ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DEL SOFTWARE

Análisis de Programas de Procesamiento de Eventos Complejos

Analysis of Complex Event Processing Programs

Realizado por
Adrián García López
Tutorizado por
Antonio Vallecillo Moreno
Lola Burgueño Caballero
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DE 2018

Fecha defensa:
El Secretario del Tribunal

Resumen

El procesamiento de eventos complejos (CEP, por sus siglas en inglés: Complex Event Processing), está ganando aceptación en los entornos distribuidos de tiempo real, al proporcionar una forma rápida y eficiente de correlacionar e inferir conclusiones sobre eventos que ocurren en tiempo real. Esta tecnología tiene un amplio campo de aplicación como pueden ser el Internet de las Cosas (IoT), monitorización de sistemas o alerta de situaciones de riesgo en infraestructuras sanitarias, entre otras. La característica más importante de estos tipos de programas, es la capacidad de expresar patrones de sucesos sobre los eventos, mediante la definición de reglas. La especificación de estos tipos de patrones se realiza utilizando lenguajes de procesamiento de eventos como Esper, el cual ha sido utilizado en este proyecto. Es muy importante la correcta especificación de estos patrones ya que de ellos depende el correcto funcionamiento del sistema.

Con tal fin, se ha desarrollado una herramienta capaz de analizar dos propiedades que pueden comprobarse estáticamente en las especificación de los programas CEP basados en reglas: la aciclicidad de las dependencias entre reglas y las condiciones de carrera entre reglas. Ambas características tienen que lidiar con el carácter no determinista de los sistemas basados en reglas.

Para el desarrollo de esta herramienta se ha utilizado un enfoque MDSE (*Model-Driven Software Engineering*). Más concretamente, se ha desarrollado un *plug-in* capaz de reconocer el lenguaje Esper y obtener como salida una representación en forma de grafo dirigido para la visualización de los resultados del análisis.

Palabras clave: Procesamiento de Eventos Complejos, Esper, Ingeniería de Software Dirigida por Modelos, Xtext, Análisis Estático.

Abstract

Complex Event Processing (CEP) is having a good acceptance in distributed real-time environments since it provides a quick and efficient way to correlate and infer conclusions about events that happen in real time. This technology can be used in several areas such as Internet of Things (IoT), systems monitoring or critical situation detection in clinical environment, among others. The most important characteristic of this kind of programs is the ability to express occurrences patterns over events, by defining rules. The specification of these types of patterns is carried out by using event processing languages such as Esper. The correct specification of these patterrns is crucial because the correct behavior of the system depends on them.

To this end, we have developed a tool capable of analyzing two properties that it can be checked through the static analysis of rule-based CEP programs: pattern acyclicity and pattern race conditions. They both have to deal with the non-deterministic nature of rule-based systems.

For the development of this tool, a Model-Driven Software Engineering approach has been used. In particular, we have developed a plug-in capable of recognizing Esper language and obtain as output a directed graph representation for visualizing the result of the analysis.

Key words: Complex Event Processing, Esper, Model-Driven Software Engineering, Xtext, Static Analysis.

Índice general

Capítulo 1

Introducción

Este capítulo servirá para contextualizar el entorno de la herramienta desarrollada, mostrando pues, su motivación, objetivos del desarrollo y, por último, se verá la estructura del presente documento.

1.1. Objetivos

El objetivo fundamental del proyecto es la realización de una herramienta capaz de analizar propiedades en la especificación de los programas de Procesamiento de Eventos Complejos [Cugola and Margara, 2012], [Etzion and Niblett, 2010] basados en reglas. La herramienta se basa en un Analizador Sintáctico [Dick and Ceriel, 1990] (también conocido como *parser*) capaz de reconocer estas reglas y, posteriormente, generar conclusiones sobre ellas.

Esta memoria describe la herramienta que hemos realizado como parte del Trabajo Fin de Grado, y que permite realizar diversos tipos de análisis estáticos sobre programas de Procesamiento de Eventos Complejos (CEP por sus siglas en inglés, *Complex Event Processing*). La herramienta se ha desarrollado siguiendo las técnicas de Desarrollo de Software Dirigido por Modelos [García et al., 2013, Brambilla et al., 2017] (DSDM por sus siglas) por la versatilidad que ofrece para el desarrollo de este tipo de aplicaciones.

En cuanto a su funcionalidad, la herramienta es capaz de detectar ciclos en la especificación de un conjunto de reglas y alertar sobre éstos. En el caso de que no se detecten ciclos, se podrá hacer una comprobación de las prioridades asignadas a las reglas y generar el mismo conjunto de reglas con las prioridades atendiendo su orden topológico.

Finalmente, al tratarse de un Trabajo Fin de Grado, los objetivos que se pretendían con este trabajo también incluyen los de conocer y familiarizarse con diversos conceptos y técnicas que no se han visto en el grado, así como poner en práctica las enseñanzas recibi-

das durante la carrera. En este sentido, para el desarrollo de la herramienta se ha tenido que aprender cómo funcionan los sistemas de Procesamiento de Flujos de Información [Cugola and Margara, 2012] (más concrétamente CEP) y cómo especificar estas reglas de las que se nutre la herramienta. También se ha aprendido a utilizar *Xtext* [Xtext, 2018], una herramienta de DSDM que ha sido esencial para el desarrollo de la herramienta objetivo de este TFG. Por otro lado, se ha utilizo todos los conceptos aprendidos durante la carrera sobre Grafos [Grimaldi, 1997], ya que ha sido la forma de representar y tratar los conjuntos de reglas para su análisis.

1.2. Motivación

El presente proyecto nace de una Beca de Colaboración [Ministerio de Educación, 2018] concedida al autor de esta memoria, que ha sido prestada en el grupo de investigación Atenea [Atenea, 2018] centrado en el Modelado de Sistemas Software.

El Procesamiento de Eventos Complejos (de aquí en adelante CEP) está ganando aceptación en la mayoría de entornos distribuidos de tiempo real y en las aplicaciones de *Big Data* al ofrecer una tecnología muy eficaz para analizar y correlacionar flujos de información e inferir conclusiones sobre ellos. Estos flujos de información se componen de eventos que ocurren en tiempo real. CEP es muy útil en diversos contexto, especialmente en el ámbito del *Internet de las Cosas* [McEwen and Cassimally, 2013], donde diversas aplicaciones han de procesar y reaccionar a eventos que provienen de diversas fuentes. A diferencia de otros sistemas de procesamiento de flujos de datos, CEP permite definir eventos complejos, patrones sobre los eventos simples, identificar situaciones complejas, y responder a ellas rápidamente. Los Lenguajes de Procesamiento de Eventos son los encargados de definir los sucesos, los patrones de eventos y las reglas de generación de eventos complejos. Este proyecto se centra en el Lenguaje de Procesamiento de Eventos Esper [EsperTech, 2018] el cual se tratará en más detalle en el Capítulo 3.

Ahora bien, conforme aumenta el uso de los programas CEP para distintos tipos de aplicaciones también crece su complejidad, lo que los hace cada vez más difíciles de comprender, depurar, mantener y probar que son correctos. Cada vez es más importante comprobar la ausencia de errores semánticos en las definiciones de los patrones de eventos y validar que el comportamiento de los programas CEP es el esperado. Ésta sería la motivación principal para el desarrollo de la herramienta.

Por otro lado, dado que el autor estaba interesado en el ámbito de la Ingeniería de Software Dirigida por Modelos, se ha optó por utilizar este paradigma para el desarrollo de la herramienta.

La Ingeniería de Software Dirigida por Modelos (de aquí en adelante MBSE, pos sus siglas en inglés, *Model-Based Software Engineering*) es un paradigma de Ingeniería de Software en donde los modelos (es decir, las representaciones abstractas de los conocimientos y actividades que rigen un dominio de aplicación particular) y las transformaciones entre ellos, se convierten en las piezas claves de los procesos de ingeniería. Este nuevo paradigma está tomando cuerpo y siendo adoptado cada vez más por las empresas de desarrollo de software, pero aún necesita estudios que permitan evaluar su comportamiento frente a métodos de desarrollo de software tradicional.

1.3. Tareas a desarrollar

Tarea 1 Construir un analizador sintáctico que convierta un programa Esper en un Árbol de Sintaxis Abstracta (AST por sus siglas en inglés, *Abstract Syntax Tree*) [Dick and Ceriel, 1990]. Esta tarea incluye el diseño, desarrollo y pruebas del analizador sintáctico.

Tarea 2 Utilizando el AST resultante de de la tarea 1, generar un representación en forma de grafo de las dependencias entre reglas.

Tarea 3 Haciendo uso del resultado de la tarea 2, utilizar el grafo generado para inferir los análisis pertinentes.

Tarea 4 A partir de estos análisis, generar una vista accesible directamente desde Eclipse [Eclipse, 2018a], que muestre el grafo resultante de haberle aplicado los diferentes análisis.

Tarea 5 A partir del análisis de prioridades entre reglas (si procede), generar el mismo conjunto de reglas introduciendo las prioridades obtenidas del análisis.

1.4. Metodología y fases de trabajo

1.4.1. Metodología

Para llevar a cabo este proyecto se ha utilizado una metodología iterativa incremental basada en técnicas ágiles. Gracias a este tipo de técnicas se consigue adaptar la forma de trabajo a las condiciones del proyecto. Este tipo de técnicas son muy versátiles y se utilizan en diversos ámbitos pero son de significativo uso en el campo tecnológico. Sus

fundamentos se basan en la constante colaboración (o integración) del cliente en el proceso de desarrollo software y la rápida adaptación a los cambios que puedan surgir a lo largo del desarrollo [Gallardo, 2018]. Para este proyecto se han utilizado los conceptos de:

- **Involucrar el cliente** a través de entregas periódicas de muestras del producto software. De esta manera obtenemos una retroalimentación efectiva y constante para poder adaptar el producto a las necesidades del cliente conforme el producto va creciendo.
- **Prototipado:** para las muestras del producto software se ha utilizado la técnica del prototipado. El prototipo no es más que una representación parcial del producto software para que el cliente compruebe que, esa representación parcial, satisface sus necesidades.
- **Iteración:** es un bloque temporal donde se realiza una evolución del producto (desarrollo de una funcionalidad) [proyectosagiles.org, 2018].

De esta manera, se han realizado iteraciones de tiempo variable con un máximo de dos semanas en cada una de las tareas que componen el proyecto. En cada iteración se ha obtenido un prototipo de la herramienta para poder iterar sobre él.

Al final de cada iteración se contrastaron los resultados con los tutores, con lo que se mantuvieron reuniones de seguimiento de forma regular y periódica, y si el resultado de la iteración era satisfactorio, se establecían los objetivos de la siguiente iteración.

Debido a que el proyecto tiene una fuerte componente de investigación, esta metodología escogida (permitiendo constantes iteraciones sobre el prototipo) es la que mejor se adecua a este tipo proyectos ya que, en cada fase de desarrollo ha existido un periodo previo de investigación sobre qué tecnologías serían las más adecuadas, y de cómo sería la forma más óptima de integrarlas en cada fase.

1.4.2. Fases de trabajo

A continuación, se muestran las fases de trabajo para el desarrollo del proyecto:

Fase 1

Estudio de los lenguajes CEP, en particular Esper. Estudio de los programas CEP, en qué se basan y en qué se diferencian con otros tipos de sistemas de Procesamiento de Flujos de Información.

Fase 2

Estudio de las principales técnicas y herramientas MDE en Eclipse. El proyecto se basa en el entorno de programación Eclipse ya que la herramienta se ha pensado para ser utilizada como una extensión de éste. Además, Eclipse es, a día de hoy, una de las compañía líder en la utilización y en la generación de productos relacionados con la Ingeniería Dirigida por Modelos (MDE).

Fase 3

Estudio de la herramienta *Xtext* [Xtext, 2018] para la elaboración del analizador sintáctico. Xtext es la herramienta fundamental de este proyecto ya que nos proporciona un conjunto de herramientas para poder crear DSLs.

Fase 4

Desarrollo del analizador sintáctico, cuya finalidad es convertir programas CEP escritos en el lenguaje Esper a un modelo conforme a un metamodelo para la representación de grafos.

Fase 5

Desarrollo de la herramienta, que dado el modelo resultante de la Fase 4, y aplicando transformaciones sobre dicho modelo, pueda analizar ciertas propiedades de los programas CEP.

1.5. Estructura de la memoria

En este apartado, se muestra la estructura de la memoria y un breve resumen sobre qué temas abarca cada una de ellas.

Capítulo 2

Estado del arte. Se muestra una visión general de los dos ámbitos fundamentales de este proyecto: los programas de Procesamiento de Eventos Complejos y la Ingeniería de Software Dirigida por Modelos. Por otro lado se muestra una visión general del lenguaje de Procesamiento de Eventos utilizado para crear sistemas CEP, lenguaje Esper, y del cual la herramienta desarrollada extrae conclusiones en su especificación.

Para el primer ámbito (CEP) se mostrará su origen y evolución, así como, una serie de ejemplos de aplicación de CEP a diferentes contextos.

En el ámbito MDSE, se explicará sus fundamentos y se centrará en los Lenguajes de Dominio Específicos (DSLs), ya que es la arquitectura básica de la herramienta.

Capítulo 3

Tecnologías a utilizar. Se describen las tecnologías utilizadas para el desarrollo de la herramienta, detallando cuál es su cometido dentro de la misma.

Capítulo 4

Estructura de la herramienta. Capítulo donde se muestra la estructura en detalle de la herramienta desarrollada, explicando cada parte de la misma, con qué tecnología se relaciona y la importancia de cada una.

Capítulo 5

Validación y pruebas. En esta sección se verá los problemas encontrados en la especificación de la gramática, cómo se ha podido depurar y cómo se ha podido corregir. Por otro lado se mostrará el método por el cual se ha validado la herramienta.

Capítulo 6

Conclusiones. Como cierre final de la memoria, en este capítulo, se mostraran las conclusiones que se han obtenido en el aprendizaje de este tipo de programas (CEP), así como de la Ingeniería del Software Dirigida por Modelos, y del desarrollo de la herramienta. También se exponen las líneas de trabajo futuro que podría realizarse sobre el proyecto para poder ampliarlo.

Capítulo 2

Estado del arte

En este capítulo sirve como presentación de los dos ámbitos de este proyecto: la Ingeniería de Software Dirigida por Modelos y los programas de Procesamiento de Eventos Complejos.

Sobre la Ingeniería de Software Dirigida por Modelos se mostrará una introducción a modo de motivación, los principios básicos sobre lo que se basa este paradigma y, para finalizar, se muestra una sección donde se entrará en mas detalle en los Lenguajes Específicos de Dominio.

En la siguiente sección se habla de los programas CEP con un enfoque *top-down*, es decir, partiendo de los conceptos generales de donde provienen este tipo de programas (*Information Flow Processing domain*), se hará una descripción de los diferentes tipos de sistemas que existen en este ámbito para, posteriormente, centrarnos de una manera más detallada en los programas de Procesamiento de Eventos Complejos y sus particularidades.

2.1. Ingeniería del Software Dirigida por Modelos

2.1.1. Introducción

El Desarrollo de Software Dirigido por Modelos (DSDM) es un paradigma de desarrollo de software que se centra en los modelos (representaciones abstracta de un sistema o de una parte de él) y las transformaciones entre ellos para realizar la mayoría de procesos de ingeniería.

De igual manera que la aparición de los lenguajes de programación supuso un salto contundente en la manera de desarrollar software, nos permiten elevar el nivel de abstracción al no tener que lidiar con conceptos sobre el nivel máquina, la Ingeniería del Software Dirigida por Modelos nace para elevar el nivel de abstracción sobre los len-

guajes de programación. Con esto conseguimos poder razonar sobre el sistema dejando de lado detalles de implementación y, a su vez, conseguir un alto nivel de automatización al poder generar partes del sistema aplicando transformaciones sobre los modelos que lo componen. Todo lo dicho anteriormente sería los objetivos fundamentales de este paradigma para combatir el principal problema del desarrollo software, su complejidad [García et al., 2013, Vallecillo, 2018].

Actualmente nos encontramos en un momento donde la Ingeniería del Software Dirigida por Modelos se encuentra a caballo entre su adopción en la industria y su validación como paradigma de desarrollo software a través de estudios que evalúen su comportamiento frente a los métodos de desarrollo software tradicionales. Para llevar a cabo una adopción total de este paradigma en la industria, podemos ver los diferentes puntos que el autor *Bran Selic* nos muestra en su artículo "Manifestaciones sobre MDA" [Selic, 2008]:

1. Completar los estudios teórico y desarrollar herramientas que sean robustas y usables.

2. Hacer ver a las empresas los beneficios de la implantación de este paradigma en sus desarrollos.

3. Disponer de personal cualificado que entienda el paradigma MDSE. Para esto hay que actuar en tres ámbitos: la investigación y desarrollo de nuevas herramientas, la enseñanza de este paradigma en las universidades y la realización y transferencia de proyectos que permitan transmitir la información obtenida en las empresas.

La Ingeniería de Software Dirigida por Modelos tuvo su gran auge con el nacimiento de UML (*Unified Modeling Language*) [OMG, 2018], una notación que nos permite especificar, visualizar y documentar los diferentes modelos de un sistema software. El problema residía en la utilización de UML para documentar al final del desarrollo, lo cual no es una acción errónea, pero haciendo esto, no se estaba sacando el máximo provecho de los modelos [García et al., 2013]. En el artículo de *Grady Booch* "*Growing the UML*" [Booch, 2002], podemos encontrar los 4 pilares fundamentales sobre el potencial de los modelos:

1. **Documentar** el proceso de desarrollo software.

2. **Razonar** sobre el propio sistema.

3. **Comunicar** ideas y fomentar la discusión sobre los diferentes aspectos del sistema.

4. **Generar** partes del sistema transformando estos modelos.

La MDSE nace para facilitar la automatización del desarrollo de software en la industria pero todavía queda un largo camino hasta que su adopción sea completa. Por

un lado, hacen falta más estudios sobre la forma de enseñar este nuevo paradigma en las aulas. En el artículo de Jordi Cabot y *Dimitris Kolovos "Human factors in the adoption of model-driven engineering: an educator's perspective"* [Cabot and Kolovos, 2016] podemos encontrar un análisis sobre los problemas encontrados en la enseñanza de este paradigma a estudiantes. Por otro lado, hacen falta integrar herramientas MDE en el ámbito específico de compañías para hacer ver el potencial la Ingeniería de Software Dirigida por Modelos. No obstante, estamos ante el paradigma de desarrollo de software que se asentará y será utilizado en el futuro.

2.1.2. Principios Básicos

La Ingeniería de Software Dirigida por Modelos no es solo un paradigma de desarrollo de software, sino que se compone de tres grandes subconjuntos los cuales podríamos definir como "subparadigmas" enfocados en un ámbito concreto [García et al., 2013].

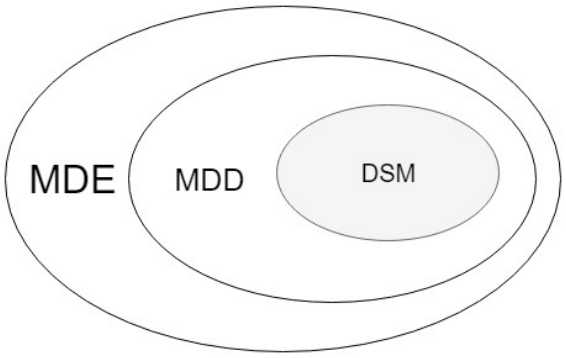


Figura 2.1: Paradigmas en los que se divida MDE y los cuales están relacionados con el proyecto.

En la Figura 2.1 podemos observar los "subparadigmas" de interés para este proyecto. MDD (por sus siglas en inglés *Model-Driven Development*) utiliza los modelos para la generación directa de artefactos [Brambilla et al., 2017]. Normalmente el resultado es fruto de la generación automática desde un modelo. Cabe destacar que existen dos "subparadigmas" más dentro de MDE (MDR y *Models@runtime*), los cuales, simplemente, se mencionan ya que se salen del conocimiento de este proyecto.

Aunque el objetivo de estos "subparadigmas" derivados de la Ingeniería Basada en Modelos (MDE) estén claramente diferenciados, todos ellos tienen los mismo principios fundamentales [García et al., 2013]:

1. Un modelo representa total o parcialmente una característica de un sistema software.

2. Estos modelos se representan con Lenguajes Específicos de Dominio (DSL).

3. Un DSL es representado a través de un metamodelo

4. Normalmente, la automatización se consigue realizando transformaciones desde los modelos a código.

Este proyecto reside en el ámbito de MDD ya que la el fin principal es la realización de una herramienta (aplicación). Más concretamente en el Desarrollo Específico del Dominio (DSM por sus siglas en inglés, *Domain-Specific Modeling*), el cual se basa en el desarrollo de DSL para intentar salvar la complejidad de un dominio específico y poder programar cercano a él. En la siguiente sección se entrará en más detalle en el campo de los Lenguajes Específicos de Dominio.

2.1.3. Lenguajes Específicos de Dominio (DSLs)

Los Lenguajes Específicos de Dominio, a diferencia de los lenguajes de programación de propósito general, están desarrollados con el fin de resolver problemas y crear construcciones específicas sobre un determinado dominio. Este dominio puede ser técnico (desarrollo de un DSL para cierto *framework*, por ejemplo) o no (área de negocio). En cualquier caso, utilizando DSL conseguimos una mayor productividad y calidad al abstraer detalles que son específicos de un lenguaje de programación (o *framework*) o de áreas de negocio [García et al., 2013].

En la Figura 2.2 podemos ver las diferentes parets de un DSL y como se relacionan entre ellas:

- **Sintaxis abstracta:** define la estructura lógica, qué expresiones son correctas utilizando los conceptos del lenguaje. En pocas palabras, nos dice cuándo un modelo está bien formado. La sintaxis abstracta se define mediante un metamodelo, que no es más que el modelo de la propia sintaxis abstracta de un DSL.

- **Sintaxis concreta:** dota al DSL de notación y aspectos de visualización. Podemos diferenciar dos tipos de sintaxis concreta: textual, las cuales son más expresivas y normalmente suelen estar basadas en gramáticas (este es el caso de *Xtext*, herramienta centrada en el desarrollo de DSL textuales. Más adelante se entrará en detalles sobre esta herramienta) y gráficos, centrados en en la representación del DSL mediante diagramas. Un mismo DSL puede tener varias representaciones (sintaxis concreta), la separación de los tipos de sintaxis (abstracta y concreta) es una característica fundamental en la Ingeniería de Software Dirigida por Modelos. Este proyecto combina aspectos de los dos tipos de sintaxis concreta de DSL, por un lado la parte de la definición de la gramática y por otro lado la visualización gráfica del DSL.

- **Semántica:** define la utilidad del DLS, es decir, la transformación de los conceptos recogidos en el DSL a conceptos cuya semántica ya es conocida.

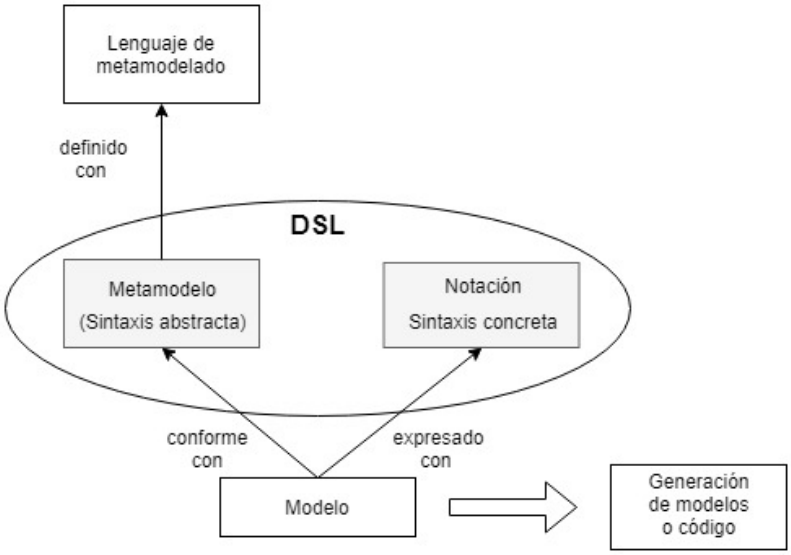


Figura 2.2: Esquema de la estructura de un DSL basado en [García et al., 2013]

Un ejemplo de Lenguajes Específicos de Dominio sería SQL [Microsoft, 2018], que no es más que un DSL para la gestión y manipulación de bases de datos, abstrayendo todos los detalles de bajo nivel. Otros ejemplos serían *CSS* (*Cascading Style Sheets*) [Mozilla, 2018] para dotar de estilo a las páginas webs y el propio lenguaje DOT [Graphviz, 2018b] utilizado en este proyecto para la representación de grafos (más detalles en el Capítulo 3).

Para concluir este apartado hay que notar que la utilización de de los DSL nos permite un aumento de la abstracción, pero esta abstracción tiene que ser seguida por un aumento de la automatización para que el uso de los DSLs sea efectivo. Esto se consigue gracias a la semántica dotada al DSL que en la mayoría de los casos son transformaciones de modelo a texto (o código) (*M2T*) y de modelo a modelo (*M2M*).

2.2. Procesamiento de Eventos Complejos

2.2.1. Information Flow Processing (IFP) domain

La forma de tratar los datos en los sistemas actuales está cambiando. Clásicamente, nos encontrábamos con sistemas en los que la información tenía que ser previamente persistida para luego ser tratada, hablamos de los *Data Base Management Systems* (DBMS). Estos sistemas se caracterizan por la poca tasa de actualización de los sistemas, es decir, el procesado de lo datos (una vez persistidos) se limita ha hacerse bajo demanda del usuario.

Cuando el usuario realiza una consulta a un DBMS (por ejemplo, una aplicación web cuyo sistema de persistencia es una base de datos relacional), esta consulta produce una respuesta que es devuelta al usuario. Podemos resumir que para cada consulta se produce un único tratamiento de los datos y el resultado es devuelto a las capas exteriores del sistema.

Hoy en día, debido a la evolución de los sistemas que necesitan nutrirse de información de una manera continua (inferir conclusiones sobre una gran cantidad de datos entrantes, alerta de situaciones críticas en el sistema, supervisión de sistemas, etc) nacen los sistemas IFP. Estos sistemas permiten el procesado continuo de datos procedentes de diversas fuentes externas al sistema. Una característica esencial de estos sistemas es la posibilidad de procesas flujos de datos sin tener la necesidad de hacer una persistencia previa de ellos. No obstante, podemos encontrar sistemas en los que sí se realicen persistencia de los datos. Para tal cometido, las arquitecturas de estos sistemas, así como sus mecanismos de procesado y sus modelos datos difieren de los tradicionales DBMS [Cugola and Margara, 2012].

El objetivo de estos sistemas es la realización de una serie de tratamientos o transformaciones de los datos para obtener conclusiones tan pronto como los datos entran en el sistema. Actualmente nos encontramos con dos tipos de sistemas IFP que son los predominantes en la industria: los sistemas de *Data Stream Processing* y los sistemas de Procesamiento de Eventos Complejos (CEP).

Los sistemas de *Data Stream Processing* nacen a raíz de una evolución de los sistemas tradicionales DBMS para dar pie a los DSMS (por sus siglas en inglés *Data Stream Management Systems*). Se basan en la transformación de flujos de datos que provienen desde fuera del sistema, para producir nuevo flujos de datos que serán tratados nuevamente (dentro del sistema) o enviados fuera de sistema. La diferencia más sustancial con los DBMS es que cuando éstos para cada consulta producen una respuesta, los DSMS mantienen un conjunto de consultas (o transformaciones) que son aplicadas a cada dato que entra en el sistema, con independencia de si son persistidos o no posteriormente.

Por su parte, el Procesamiento de Eventos Complejos se basa en el tratamiento de los datos como notificaciones de eventos que se introducen en el sistema o se producen dentro del mismo. Tiene sus orígenes en el modelo Publicador-Suscriptor [Eugster et al., 2003] salvando la diferencia de que en el modelo Publicador-Suscriptor, los eventos son considerados aislados unos de otros (se tratan individualmente) y en los sistemas CEP podemos inferir conclusiones cuando un determinado patrón de eventos ocurre en el sistema.

2.2.2. Sistemas de Procesamiento de Eventos Complejos

Los sistemas CEP tratan los datos como notificaciones de eventos. Para saber a qué nos referimos con el concepto de evento podemos irnos a la definición que se da en [Etzion and Niblett, 2010]: "Un evento es una ocurrencia dentro de un sistema o dominio particular; es algo que ha sucedido, o se contempla como ocurrido en ese dominio. La palabra evento también se utiliza para definir una entidad de programación que representa la ocurrencia de tal suceso en un sistema informático". En los sistemas CEP utilizamos la segunda definición de evento, una representación informática de una ocurrencia. Necesitamos tal definición porque, como se ha comentado con anterioridad, la principal característica de CEP es la habilidad de poder reaccionar cuando una serie de eventos concretos ocurra en el sistema, y para ello, necesitamos tal representación informática para poder tratarlo.

En los sistemas CEP podemos definir dos tipos de eventos:

- **Eventos simples:** normalmente estos eventos son generados mediante los productores de eventos. Éstos pueden ser: un sensor (por ejemplo de temperatura), un proceso de negocio (por ejemplo, un proceso de reserva de habitaciones de un hotel que al final del mismo emite un evento de habitación reservada), un sistema (el cual detecta una sobrecarga de tráfico en la red y lo notifica mediante el envío de un evento), etc. Los diferentes productores enviaran los eventos generados a un sistema de procesado (en este caso un motor de Procesamiento de Eventos Complejos) para su tratamiento.

- **Eventos complejos:** en un sistema CEP, para poder reaccionar a determinados patrones de eventos, tenemos que definir reglas que expresen ese patrón. Por ejemplo: si tenemos un sistema que recibe eventos de temperatura y nuestro motor tiene una regla que, cuando recibe una evento de temperatura cuya temperatura es mayor que 80^{o}, produce un evento de encendido de los rociadores de agua, el evento resultante de ejecutar esta regla sería considerado como un evento complejo. Los eventos complejos pueden ser enviados directamente a los consumidores de eventos, que no son más que sistemas que reciben estos eventos complejos y realizan una cierta acción en consecuencia. Pueden ser desde sistemas de persistencia, actuadores o otros procesos de negocio. También, estos eventos complejos pueden ser consumidos nuevamente por otras reglas del motor de procesamiento CEP, si hay alguna regla que requiera de la presencia de algún otro evento complejo para ser ejecutada.

Las reglas mencionadas anteriormente son especificadas a través de lenguajes de Procesamientos de Eventos y tienen una estructura en común en la mayoría de los sistmas CEP [Moreno et al., 2018]:

- **Fase de selección:** en esta fase se analiza cuáles son los eventos, tanto simple como complejos, que hacen que la regla se ejecute. Para la única regla existente en el ejemplo anterior del sensor de temperatura, en esta fase tendríamos que la regla solo se ejecutaría con eventos de tipo temperatura. En resumen, para cada regla tendremos, tras esta fase, lo que denominaremos el conjunto de dependencias, ya que en él estarán todos los eventos de los que depende la ejecución de la misma.

- **Fase de emparejamiento:** teniendo en cuanta los eventos de la fase de selección, tenemos que ver si todos ellos cumplen los requisitos para la ejecución. Volviendo al ejemplo anterior, la regla solo se ejecutaría si el evento de tipo temperatura contiene una temperatura mayor a 80^{o}. Este es un ejemplo muy simple, pero en escenarios más complejos se podría combinar los eventos de la fase de selección con operadores lógicos (`and`, `or`, `->`, etc.).

- **Fase de producción:** en esta fase se definen qué tipo de datos, extraídos de los atributos de los eventos de la fase de selección, se van a producir en forma de evento en el caso de que la regla se ejecute. En el ejemplo anterior, cuando enviamos el evento de encendido de los rociadores, podríamos enviar como atributos un identificador del sensor que ha producido la alerta, para que el consumidor sepa dónde se ha producido la alerta.

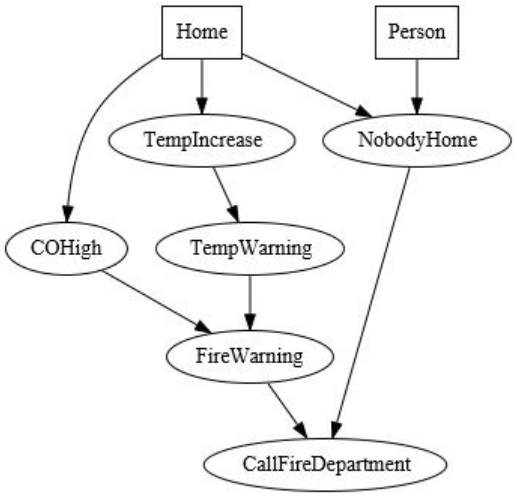


Figura 2.3: Grafo dirigido de las dependencias entre reglas extraído de [Moreno et al., 2018]

2.2.3. Análisis estático de programas CEP

Los programas CEP son programas basados en reglas para poder inferir conclusiones mediante la definición de ciertos patrones de eventos. A raíz de lo anterior comentado, surgen dos propiedades características de estos tipos de programas: la aciclicidad y el orden

entre reglas. Estas propiedades ocurren debido al carácter no determinista y confluente (el orden de ejecución de las reglas importa) de estos sistemas [Burgueño et al., 2018]. Estas dos propiedades son las que se han conseguido automatizar mediante el desarrollo de la herramienta.

Aciclicidad de las reglas

Supongamos que tenemos las reglas que están representadas en la Figura 2.3 como un grafo dirigido, donde los nodos con forma de rectángulos son eventos simples y los eventos con forma de óvalo son eventos complejos. Este grafo muestra sistema CEP en una *Smart House* y no tiene ningún ciclo entre sus eventos.

Imaginemos que, por error, la regla *"COHigh"* tuviera en su conjunto de dependencias al evento *"FireWarning"*, con lo cual se estaría formando un ciclo entre ambas reglas. Esto no quiere decir que la especificación de las reglas sea incorrecta, pero puede llegar a producir situaciones de bucles infinitos (una regla que produce un tipo de evento que consume otra regla que, a su vez, produce un evento que consume la primera). Alertar de esta situación puede suponer corregir errores críticos en la especificación.

Orden entre reglas

Dadas dos reglas R_1, R_2, la primera consume eventos de tipo b y produce eventos de tipo a, y la segunda consume eventos de tipo c y produce eventos de tipo b. Supongamos que recibimos un evento de tipo c y que R_2 se ejecuta antes que R_1, con lo cual obtendrías como resultado la salida de dos eventos, uno de tipo b y otro de tipo a. El problema surge cuando primero se comprueba la regla R_1 y, posteriormente, R_2. A la llegada del evento c la regla R_1 no produciría nada ya que no consume eventos de este tipo, pero la regla R_2 sí que lo haría, con lo cual obtendríamos como resultado un evento de tipo b.

Para solventar este tipo de situaciones algunos lenguajes de Procesamiento de Eventos incluyen la posibilidad de incluir prioridades a las reglas, de tal manera que, a la llegada de un evento, siempre se comprueben antes las más prioritarias. En el ejemplo anterior, R_2 sería más prioritaria que R_1.

2.3. Esper

Esper [EsperTech, 2018] es un motor de procesamiento de eventos complejos de código abierto (*open-source*) que pertenece a la compañía EsperTech Inc. Proporciona un lenguaje de procesamiento de eventos para especificar reglas sobre eventos simples y compuestos.

Como se ha comentado en la sección anterior, este tipo de motores son capaces de correlacionar considerables cantidades de eventos en un tipo ínfimo. Esto es posible, gracias al cambio de en la manera de tratar los datos (o eventos). En la Figura 2.4 podemos ver en un enfoque (clásico) donde los datos son los persistidos y sobre ellos se lanzan consultas y, por otro lado, el enfoque en el cual se basan los sistemas CEP, en este caso, los patrones (reglas) son los persistidos y sobre ellos se lanzan los datos (eventos) para producir resultados.

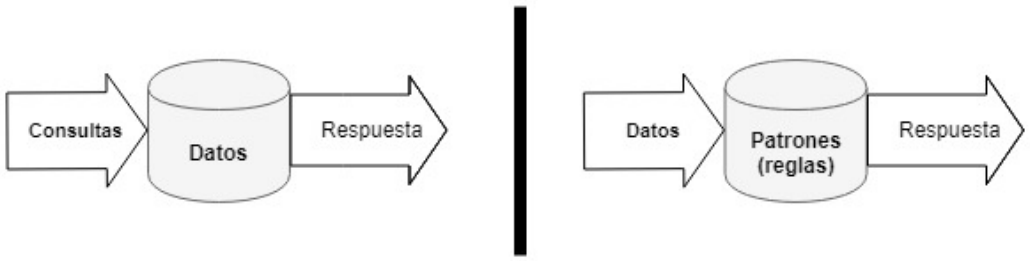


Figura 2.4: A la izquierda un enfoque estático de procesamiento de datos. A la derecha un enfoque dinámico orientado a sistemas CEP.

El objetivo de la herramienta desarrollada en este TFG es el análisis en la especificación de programas Esper, con lo cual ha sido muy importante conocer cuál es la estructura de las reglas y cómo expresar el patrón deseado. A continuación, se muestra una explicación, a partir de un ejemplo, de todos los elementos que el analizador léxico implementado es capaz de reconocer.

```
create schema Motorbike(tirePressure1 Int, tirePressure2 Int, speed Double, seat Boolean);

@Name('BlowOutTire')
insert into BlowOutTire
@Priority(0)
select current_timestamp() as timestamp, a2.motorbikeId as motorbikeId,
    a1.location as location_a1,
    a1.tirePressure1 as tirePressure1_a1,
    a1.tirePressure2 as tirePressure2_a1,
    a2.location as location_a2,
    a2.tirePressure1 as tirePressure1_a2,
    a2.tirePressure2 as tirePressure2_a2
from pattern [(every a1 = Motorbike(a1.tirePressure1 >= 2.0) ->
    a2 = Motorbike(a2.tirePressure1 <= 1.2)) where timer:within(5 milliseconds)
    or
    (every a1 = Motorbike(a1.tirePressure2 >= 2.0) ->
    a2 = Motorbike(a2.tirePressure2 <= 1.2)) where timer:within(5 milliseconds)];
```

Figura 2.5: Ejemplo de la especificación de un evento simple y una regla en Esper

Como se puede observar en la Figura 2.5, la sintaxis de Esper, en la especificación de reglas, es muy parecida a la de SQL (lenguaje de consultas para base de datos).

Para crear un evento simple (primera línea del ejemplo), se hará utilizando las palabras reservadas `create schema` seguido de un nombre para ese evento simple (en este caso "Mo-

torbike"). Opcionalmente, entre paréntesis, podremos especificar los atributo que tendrá este evento. En el ejemplo, este evento, que representa un evento de tipo motocicleta, tiene como atributos la presión de las dos ruedas (un número entero), la velocidad (`double`) y un atributo de tipo `Boolean` que representa si el conductor está sentado o no en el asiento.

En la siguiente parte de la imagen, podemos observar la definición de una regla cuyo nombre se especifica con la anotación `@Name` y entre, paréntesis y comillas, el nombre (en este caso "BlowOutTire" (rueda pinchada).

Un elemento muy importante a la hora de especificar las reglas, es hacerla visible a todas las demás. Es decir, expresar qué tipo de evento complejo va a generar tras su ejecución. Esto se consigue con la sentencia `insert into` seguida del nombre del evento complejo. Si se omitiera esta sentencia la regla no estaría visible al resto y el evento complejo generado por este patrón no estaría disponible para su uso en otro. Es posible que varias reglas produzcan el mismo tipo de evento complejo (todas ellas tendrías en mismo nombre en la sentencia `insert into`).

La notación `@Priority` sirve para asignar prioridad a la regla. Si la prioridad se omite, se asocia la máxima prioridad por defecto (prioridad 0). Como se comentó en la sección anterior, es muy importante asociar prioridades correctas a las reglas en función a sus dependencias, para respetar el orden de ejecución de las mismas y no perder información (eventos) generados por otras reglas.

La parte más importante en la definición de una regla, es la especificación del patrón de eventos que debe darse para que ésta se ejecute. Para ello, podemos observar en la Figura 2.5 la sección `from`. En esta parte de la regla encontraremos la fase de selección (donde se encuentran el conjunto de dependencias de la regla) y la fase de emparejamiento (criterios que deben ocurrir en los atributos de los evento del conjunto de dependencias de la regla para su ejecución). Esta parte es la más crítica en el herramienta y donde más se ha indagado en su desarrollo, debido a que, de ella se extrae toda la información necesaria para analizar las dos propiedades (aciclicidad y orden de ejecución) mencionadas con anterioridad. Para describir el patrón se utilizada la palabra reservada `pattern` y entre corchetes se expresa el patrón.

Para poder expresar patrones de eventos, a continuación se explican los operadores más relevantes en Esper para su definición:

Followed-By

Este operador, que se denota con el símbolo `->`, se utiliza para especificar ordenes de precedencia entre reglas. De tal forma que solo cuando la parte izquierda del operador sea verdad, la parte derecha estará disponible para ser emparejada con los eventos entrantes.

Por ejemplo, si en la definición del patrón de una regla tenemos una expresión `A->B`, esta regla solo se ejecutará cuando primero se encuentre un evento de tipo A y, posteriormente, un evento de tipo B. También podemos añadir condiciones adicionales en el orden de precedencia, por ejemplo, haciendo referencia al ejemplo de la Figura 2.5, tenemos la siguiente expresión: `a1 = Motorbike(a1.tirePressure1 >= 2.0) ->a2 = Motorbike(a2.tirePressure1 <= 1.2)`, donde tanto `a1` como `a2` son asignaciones de los eventos para poder utilizarlas en cualquier parte de la regla. Para que la regla se ejecute tiene que recibir un evento de tipo "Motorbike" pero, además, el atributo "tirePressure1" tiene que ser mayor o igual a 2; y una vez se detecta la aparición este evento tendrá que recibir otro evento "Motorbike" pero, esta vez, el atributo "tirePressure1" tiene que ser menor que 1.2. Si ocurre todo lo descrito se activaría la regla.

De una manera análoga se podría utilizar el operador *Followed-By* para establecer patrones con más de dos reglas, es decir, `A->B->C`.

Operador "Every"

El operador *Every*, notado con la palabra reservada `every`, hace posible que el patrón al cual es aplicado, se siga comprobando una vez éste ya ha sido comprobado. Es decir, el patrón de una posible regla podría ser un simple evento A, entonces, la regla se ejecutaría cuando ocurriera un evento de tipo A. Ahora bien, si tuviéramos un patrón como `every A`, esta regla siempre se quedaría esperando eventos de tipo A aunque ocurriera algún evento de este tipo. Este operador crea un "subpatrón", que espera las mismas condiciones del patrón original al que se le aplica cada vez que ocurre el patrón de eventos especificado.

Considerando esta secuencia de eventos A_1, B_1, C_1, B_2, A_2, D_1, A_3, B_3, E_1, A_4, F_1, B_4, podemos ver una serie de ejemplos que combinan los dos operadores anteriores:

- `every (A -> B)`: este patrón detectaría todos los eventos A que son seguidos de un evento B. Cuando el patrón coincide con una secuencia de eventos entrantes, se vuelve a reiniciar para seguir buscando ese tipo de patrón. Como resultado de aplicar este patrón a la secuencia anterior, obtendríamos las siguientes parejas de eventos para los cuales se ejecutaría la regla: $\{A_1, B_1\}$, $\{A_2, B_3\}$, $\{A_4, B_4\}$.
- `every A -> B`: en esta regla el operador *every* solo se aplica a la ocurrencia de A, con

lo cual, cada vez que ocurre un evento de tipo A, se volverá a lazar un "subpatrón" similar a éste, en búsqueda de un evento de tipo B. El resultado de aplicar este patrón a la secuencia de evento sería igual al anterior añadiendo el par $\{A_3, B_3\}$,

- `A -> every B`: este patrón se ejecutará cuando ocurra un evento A y, después, ocurra un evento B; pero al estar aplicado el operador *every* al evento B, seguirá buscando ocurrencias de este tipo de evento y para cada una de ellas se ejecutará la regla. Este patrón se ejecutaría con las siguientes secuencia de eventos: $\{A_1, B_1\}$, $\{A_1, B_2\}$, $\{A_1, B_3\}$, $\{A_1, B_4\}$.

- `every A -> every B`: este patrón se ejecutará para cualquier secuencia de un evento A que le siga un evento B, debido a que el operador *every* está aplicado a ambos lados del operador *followed-By*. Este patrón se activará con la llegada de los siguientes eventos: $\{A_1, B_1\}$, $\{A_1, B_2\}$, $\{A_1, B_3\}$, $\{A_1, B_4\}$, $\{A_2, B_3\}$, $\{A_2, B_4\}$, $\{A_3, B_3\}$, $\{A_3, B_4\}$, $\{A_4, B_4\}$.

Estos patrones se pueden combinar con operadores lógicos (`and`, `or`), como se observa en la Figura 2.5, de tal forma que el patrón se ejecute como si se estuviera evaluando una expresión lógica, pero las expresiones serían, a su vez, otros patrones.

Por último, continuando con el ejemplo de la Figura 2.5, la fase de producción se encuentra en la sección `select` de la regla. En esta sección se exponen qué atributos serán generados cuando la regla se ejecute, en forma de evento complejo (estos atributos también pueden ser utilizados dentro de la misma regla). Para ello, se puede utilizar el identificador de cada evento en la sección `from` de la regla y acceder a sus atributos, así como, asignarle a este nuevo atributo un identificador para que otras reglas puedan acceder a él, utilizando la palabra reservada `as`. Cabe destacar que también se podrían utilizar funciones predefinidas de Esper (suma, resta, calculo de medias aritméticas, etc.) para producir sus datos resultantes. En el ejemplo, se utilizada la función `current_timestamp()` que devuelve la hora, en milisegundos, del sistema y que será accesible por otras reglas a través del identificador "timestamp".

Capítulo 3

Tecnologías utilizadas

3.1. Java

Java es un lenguaje de propósito general orientado a objetos. Este lenguaje se diseñó con la filosofía *"write once, run anywhere"*, es decir, poder desarrollar código Java sin tener que preocuparnos en qué plataforma lo estamos haciendo. Con tal fin, se creó la JVM (*Java Virtual Machine*), de tal forma que cuando vamos a compilar nuestro código Java (fichero con extensión `.java`), generamos un archivo intermedio con extensión `.class` (*bytecode*), que es ejecutado por cualquier implementación de la JVM.

3.1.1. Librerías y framework utilizados

En el desarrollo de la herramienta se pueden destacar por su frecuencia de uso, la API (por sus siglas en inglés, *Application Programming Interface*) de *Stream* y *Eclipse Modeling Framework* (EMF) [Eclipse, 2018b].

- **Stream API:** Debido a que la herramienta hace un uso intensivo de colecciones (diccionarios, listas, pilas y conjuntos), se ha utilizado esta API para facilitar operaciones tales como filtrado, modificación y combinación de las colecciones.

  Un *stream* representa una secuencia de datos a la cual se le aplican una serie de operaciones a cada elemento uno por uno. La colección a la cual se le aplica el *stream* no es modificada por ninguna operación que se realice sobre sus elementos, simplemente, éstos son enviados uno por uno al *stream*, donde se generan nuevos elementos que son el resultados de aplicarles estas operaciones. Esta forma de tratar las colecciones proporcionar una manera mucho más expresiva y mantenible de poder realizar operaciones sobre colecciones. En la Figura 3.1 se puede ver un ejemplo de su implementación, la colección entrante es un diccionario y se le aplica a cada elemento un filtrado (`.filter`) y la operación `map` que aplica la función que se le pasa por parámetro a cada elemento resultante del filtrado. Como se ha mencionado

con anterioridad, este conjunto de operaciones no modifican la colección entrante, con lo cual, si queremos que el resultado de las operaciones se devuelva en una colección, tenemos que hacerlo explícitamente mediante la operación `collect`.

```
mapRulePartsToString.entrySet
    .stream
    .filter[entry | entry.key.name.equals(insertName)]
    .map[entry | entry.value]
    .collect(Collectors.toList())
```

Figura 3.1: Ejemplo del filtrado de un diccionario utilizando la API de *streams*.

- **EMF:** según la web oficial de EMF [Eclipse, 2018b], podemos definirlo como: "un *framework* de modelado para facilitar la creación de herramientas de generación de código y aplicaciones basadas en un modelo de datos estructurado". Este *framework* se ha utilizado ya integrado dentro de la herramienta Xtext (a continuación descrita) y nos ha permitido la transformación de la gramática (gramática del lenguaje Esper) del DSL a clases Java para poder realizar los análisis de las propiedades descritas en el capítulo anterior.

  Otra característica de este *framework* utilizada en este proyecto ha sido la posibilidad de crear instancias en tiempo de ejecución del propio IDE (Entorno de Desarrollo Integrado), para poder ejecutar en él todo el analizador sintáctico y el generador de código.

3.2. Xtend

Xtend [Eclipse, 2018d] es un lenguaje de propósito general muy parecido sintácticamente a Java, de hecho, se basa en la JVM y el código generado tras la compilación es código Java. Este lenguaje nace para mejorar algunas características del lenguaje Java. Entre las características principales de este lenguaje podríamos destacar: la inferencia de tipos, la posibilidad de generar plantillas para la generación de código o ficheros y la utilización de algunas características de la Programación Funcional como las expresiones *lambda*.

Xtend es totalmente interoperable con cualquier librería escrita en Java. Es decir, podemos utilizar en nuestro código Xtend implementaciones escritas en código Java y, de igual manera, podemos acceder desde código Java a cualquier funcionalidad en escrita Xtend. En este proyecto se ha utilizado Xtend para el tratamiento inicial de los datos y la generación de código, y Java para aplicar los análisis a estos datos.

3.3. Xtext

Xtext [Xtext, 2018] es un framework *open-source* para crear lenguajes textuales. Actualmente, el desarrollo de este framework está liderado por la empresa Itemis AG. Xtext se instala en el IDE Eclipse como un *plug-in* que está basado en EMF, con lo cual, esto lo hace interoperable con otros *plug-ins* de Eclipse basados también en EMF.

En este proyecto, Xtext es una pieza crucial, ya que nos proporciona toda la arquitectura necesaria para realizar tanto el analizador léxico (también conocido como *lexer*) como el analizador sintáctico (*parser*).

El primer paso para realizar un DSL, es la especificación de la gramática (sintaxis abstracta). Para ello Xtext nos proporciona dos métodos:

1. **Desde cero:** el diseñador tendrá que realizar la gramática desde cero, es decir, tendrá que escribir las reglas para la correcta definición de la gramática. Cuando se tiene realizada la gramática, Xtext generará un metamodelo EMF a partir de ésta, ya que lo necesita para su correcto funcionamiento.

2. **Desde un metamodelo EMF:** este método es la inversa del método anterior. Por lo tanto, partiendo de un metamodelo EMF, *Xtext* genera la gramática (o un esqueleto de la misma), dependiendo del nivel de detalle del metamodelo.

En este TFG, debido a que la gramática de la herramienta emula la gramática del lenguaje de Procesamiento de Evento Esper (de una manera simplificada), en este proyecto se ha especificado la gramática desde cero para tener más control sobre la complejidad de las reglas y aspectos de recursividad en la gramática.

Una vez definida la gramática, Xtext genera una serie de *plug-ins* tales como un analizador sintáctico que nos permitirá generar código y un editor gráfico para poder escribir el DSL. Este editor cuenta con características como coloración de la sintaxis, autocompleción del lenguaje y detección de errores. Todas estas características (menos la coloración) vienen deshabilitadas por defecto y corren a cargo del usuario en su implementación.

3.3.1. Estructura de un proyecto Xtext

Para crear un proyecto Xtext, una vez se tenga el *plug-in* instalado, habrá que seleccionar *File* →*New* →*Project* →*Xtext project*, dentro de Eclipse. Tendremos que introducir un nombre para el proyecto y una extensión para el DSL (en este caso ha sido `.mydsl`).

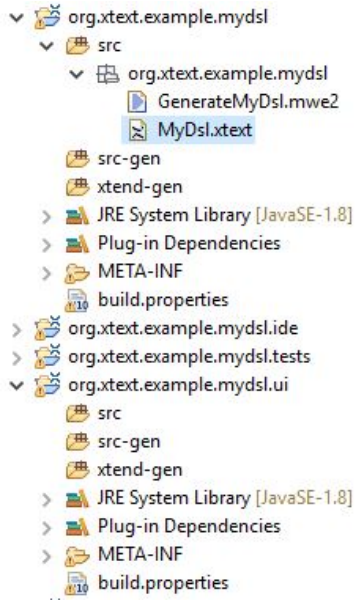


Figura 3.2: Jerarquía de directorios de un proyecto Xtext.

En la Figura 3.2 se muestra la jerarquía de subproyectos generados al crear el proyecto. En el proyecto *org.xtext.example.mydsl* nos encontramos el fichero donde expresaremos nuestra gramática (*MyDsl.xtext*) y el archivo *GenerateMyDsl.mwe2* donde se encuentran los flujos de tareas (*workflow*) que permitirán crear el editor y poder generar trazas de la gramática para su depuración. Cuando se genera la gramática a partir del archivo de flujos de tareas, se utiliza el generador ANTLR, si no se tiene previamente instalado, la herramienta solicitará la descarga. Xtext utiliza este generador para crear el *parser*.

Cuando se genere el código del editor y del *parser* estos serán depositados en la carpeta *src-gen* de los proyectos *org.xtext.example.mydsl* y *org.xtext.example.mydsl.ui*. En el primero se guarda el código del *parser* y en el segundo el código del editor, es decir, los aspectos sobre la coloración, autocompleción, y comprobación de la gramática [García et al., 2013].

3.3.2. Ejemplo de uso

En la Figura 3.3 podemos observar una definición muy simple de la gramática de un DSL. Más concretamente, la gramática es una versión muy simplificada de la gramática del lenguaje Esper. Tomándolo como punto de partida, vamos a comentar los conceptos clave que Xtext nos proporciona para definir gramáticas.

1. La definición de la gramática se basa en expresar reglas que definan la sintaxis del lenguaje que se desea diseñar. El nombre de las reglas se empieza en mayúsculas y su nombre tiene que ser único.

2. La primera regla (línea 5) se considera la regla de inicio. Esta regla indica que los programas Esper están compuestos por eventos simples o reglas. Para poder expresar

```
grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals

generate myDsl "http://www.xtext.org/example/mydsl/MyDsl"

Domainmodel :
    (rules+=RuleParts | events+=Event)*;

Event:
    'create' 'schema' name=ID eventattributes=Attributes ';';

Attributes:
    '(' attribute+=AttributesDefinition (',' attribute+=AttributesDefinition)* ')'
;

AttributesDefinition:
    name+=ID type+=ID
;

RuleParts:
    (nameRule = Name) (insert = Insert) (priority = Priority)? (selectRule = Select) (fromRule = From) ';';

Insert:
    'insert' 'into' name=ID ;

Name:
    '@Name' '(' name=STRING ')';

Priority:
    '@Priority' '('priorityInt = INT')'
;

Select:
    'select' ( selectAttributes += SingleSelectDefinition ('as' alias+=ID)? )+
           (',' selectAttributes += SingleSelectDefinition ('as' alias+=ID)?)*

    | (asterisk?='*')
;

SingleSelectDefinition:
    event+=[SingleDefinition] '.' (attribute+=ID | '*' )
;

From:
    'from' singleDefinition=SingleDefinition
;

SingleDefinition:
    (name=ID '=')? simpleEvents=[Event]
;
```

Figura 3.3: Definición de una gramática en Xtext

más de una regla o evento tenemos que dotar de multiplicidad a la regla, esto se consigue utilizando los operadores de cardinalidad (+, *, ?). Para este caso, a la regla se le impuesta que puede tener cero o muchas (operador *) reglas o eventos. Esta regla hace de punto de acceso (o nodo raíz) en el árbol de sintaxis abstracta que el *parser* crea a partir del *lexer*.

3. Las reglas pueden ser asignadas mediante operadores de asignación (=, +=, ?=). Gracias a esta asignación, cuando el *parser* genere el AST, podremos acceder a ellas como si fueran atributos de una clase *Java*. En la Figura 3.3, se puede observar que en la primera regla se hace uso del operador `+=` para asignar todas las ocurrencias de reglas al atributo `rules` de la regla principal. Este atributo es una lista que contiene todas las instancias de reglas del archivo.

4. Las reglas pueden contener *keywords* representadas como caracteres entre comillas

simples.

5. A una regla se le puede asignar un atributo `name`, el cual representa un identificador para la regla. Por ejemplo, para la regla `Event` es lo lógico que su identificador sea el nombre del evento, para ello basta con asignar al atributo `name` el valor deseado.

6. En la definición de una regla podemos realizar referencias cruzadas, es decir, podemos asignar a un atributo de la regla, el valor del identificador del atributo `name` de otra regla. Para hacerlo, simplemente hay que poner el nombre de la regla entre corchetes. Como se ve en la última regla (línea 47), al atributo `simpleEvents` se le asignara el valor `name` de la regla `Event`

Si el lector está interesado en profundizar en todos los recursos que proporciona *Xtext* para la definición de la gramática, se le remite a la documentación de este *framework* [Xtext, 2018].

3.3.3. Editor de texto

Una vez creada la gramática podemos lanzar el editor de texto para comprobar la gramática definida en el apartado anterior. Para ello, habría que pulsar botón derecho sobre la venta donde se ha definido la gramática y pulsar sobre *Run As* →*Generate Xtext Artifacts*. Una vez hecho, podemos observar que *Xtext* genera para cada regla una representación de ella encapsulada en una clase *Java*, véase Figura 3.4. Así pues, cuando en el editor de texto detecte una regla, se devolverá una instancia de ésta al *parser*, y éste, generará el AST acorde al conjunto de instancias que haya en el editor.

Para abrir el editor basta pulsar botón derecho sobre el primer proyecto (en este caso *org.xtext.example.mydsl*) y pulsar *Run As* →*Eclipse Application*. Una vez hecho esto, se abrirá una segunda instancia de Eclipse. Sobre ella tendremos que crear un proyecto haciendo *File* →*New* →*Project...* →*Java Project* y dentro del proyecto crear un archivo con la extensión del DSL que se especificó en la creación del proyecto *Xtext*.

En la Figura 3.5 se puede observar la nueva instancia de Eclipse donde se despliega el editor de texto y donde se pueden escribir programas con la gramática diseñada.

3.3.4. Generación de código

Uno de los principales potenciales del uso de DSL es la capacidad de generar código a partir de ellos. *Xtext* proporciona una manera simple de realizarlo a través del lenguaje *Xtend*.

Para ello, solo ha de abrirse la clase *Xtend* “DslGenerator.xtend” situada en el paquete que termina con la extensión “.generator”. En esta clase nos encontremos lo que se

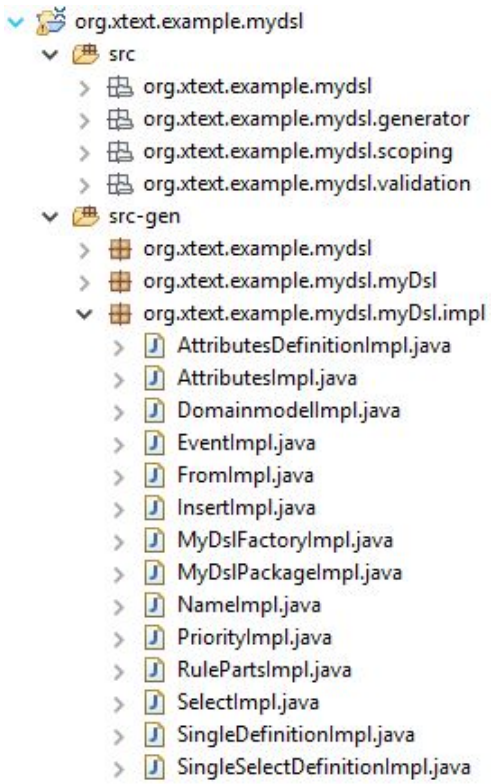


Figura 3.4: Clases generadas a partir de la gramática.

denomina una función de "*callback*" llamada "doGenerate" que, cada vez que guardemos nuestro archivo en la segunda instancia de eclipse, el motor de *Xtext* llamará a esta función pasándole tres parámetros:

1. **Resource:** este parámetro pertenece a la API de EMF, y hace posible el acceso al AST creado por el *parser* para poder navegar, a través de sus relaciones, entre las instancias creadas de las reglas definidas en la gramática.

2. **IFileSystemAccess2:** este parámetro perteneciente a la propia API de *Xtext* nos abstrae operaciones sobre ficheros, con lo cual, podremos crear, sobrescribir y modificarlos de una manera sencilla.

3. **IGeneratorContext:** interfaz del contexto del generador. No se ha requerido utilizar este parámetro en la generación.

En la Figura 3.6 podemos ver una simple implementación de este *callback* que, cada vez que guardemos el archivo de nuestro DSL en la segunda instancia de Eclipse, generará un archivo de texto plano el cual contendrá el nombre de todos los eventos simples que haya en la especificación. Se puede observar el potencial que ofrece *Xtend* al ofrecer un mecanismo para realizar plantillas de código (contenido del archivo a generar) mediante sentencias de escape denotadas por los símbolos «».

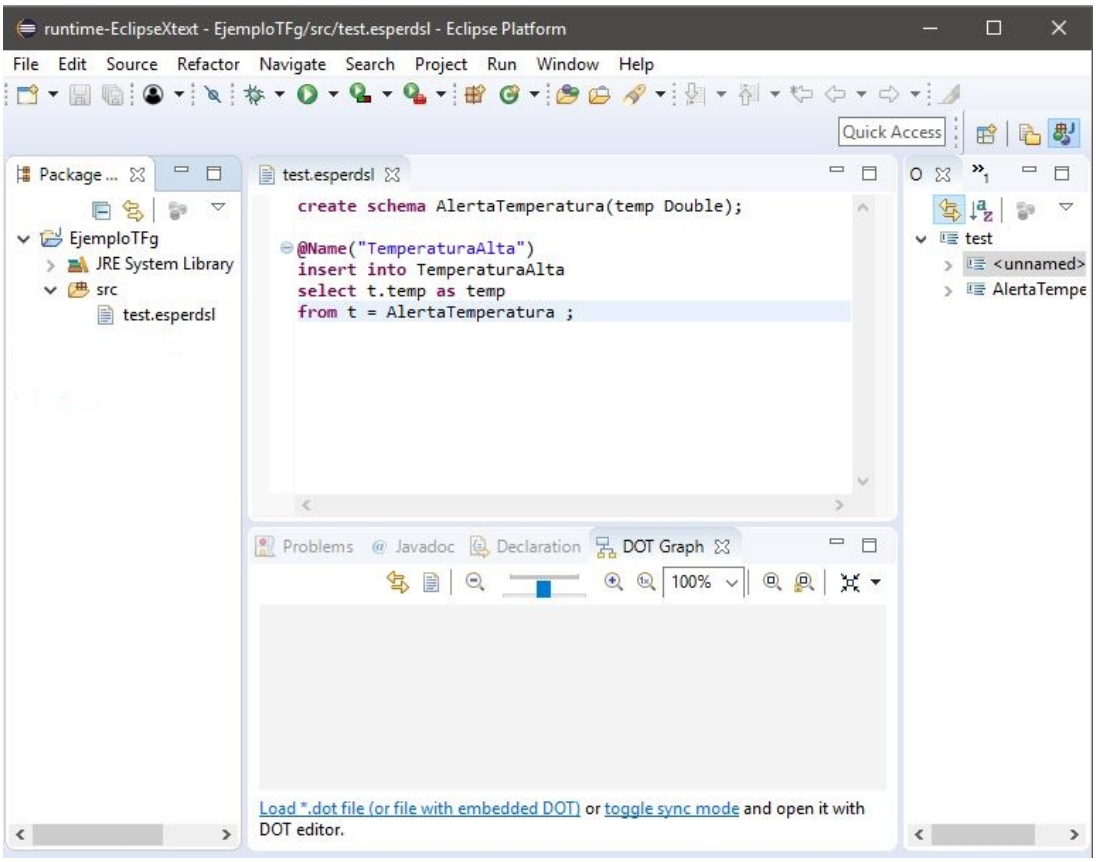


Figura 3.5: Segunda instancia de Eclipse donde se abre el editor de texto.

3.4. ANTLRWorks

ANTLRWorks [ANTLR, 2018], es un entorno de desarrollo para gramáticas ANTLR. Su uso ha sido muy específico, se ha utilizado para comprobar el correcto funcionamiento de la gramática desarrollada. A medida que ésta se hacía más compleja, podía volverse ambigua, este tipo de problemas no son fácilmente detectables y esta herramienta nos proporciona una manera sencilla de depurar gramáticas ANTLR. Un ejemplo de su uso se verá en el Capítulo 5.

3.5. GEF (Graphical Editing Framework)

GEF [Eclipse, 2018c] proporciona, de una manera integrada con el IDE Eclipse, una serie de herramientas para la visualización y desarrollo de aplicaciones gráficas. Se puede instalar como un *plugin* de una manera simple. Esta herramienta se ha utilizado para la visualización de grafos escritos en lenguaje DOT en una vista nativa en Eclipse (véase la Figura 3.7), ya que trae un interprete de este lenguaje. Su instalación se muestra en el Apéndice A.

```
2⊕ * generated by Xtext 2.13.0
4  package org.xtext.prueba.esper.generator
5
6⊕ import org.eclipse.emf.ecore.resource.Resource
11
13⊕ * Generates code from your model files on save.
17⊖ class DslGenerator extends AbstractGenerator {
18
19⊖     override void doGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context) {
20          var model = resource.contents.head as Domainmodel
21          var simpleEvents = model.events
22          fsa.generateFile('EventosSimples.txt',
23⊖             '''
24              Eventos simples:
25⊖             «FOR simpleEvent : simpleEvents»
26              «simpleEvent.name»
27              «ENDFOR»
28              '''
29          )
30      }
31  }
```

Figura 3.6: *callback* para realizar generación de códig en *Xtext*.

3.6. Graphviz

Graphviz [Graphviz, 2018b] es un software libre para la visualización de grafos. Esta herramienta coge descripciones de grafos en lenguajes como DOT y los representa de una manera más visual en formatos como PDF, SVG, formatos de imagen, etc. Para la herramienta desarrollada, la instalación de Graphviz no es obligatoria pero sí muy recomendable ya que, al hacerlo, nos permitirá desde Eclipse exportar el grafo en el formato que el usuario desee. Además, Graphviz permite una mejor visualización del grafo a través de la ventana gráfica de GEF, frente al uso del propio interprete DOT de GEF.

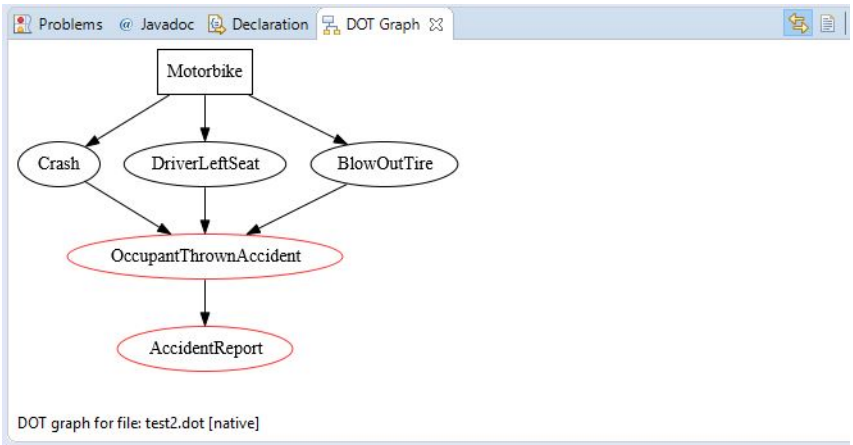


Figura 3.7: Vista nativa del interprete del lenguaje DOT proporcionado por GEF.

3.6.1. DOT

DOT es un lenguaje para especificar grafos de una forma sencilla. En la Figura 3.8 se puede observar un ejemplo de su definición. Para grafos dirigidos (únicos utilizados en este proyecto), la definición del grafo tiene que empezar con la palabra reservada `digraph`

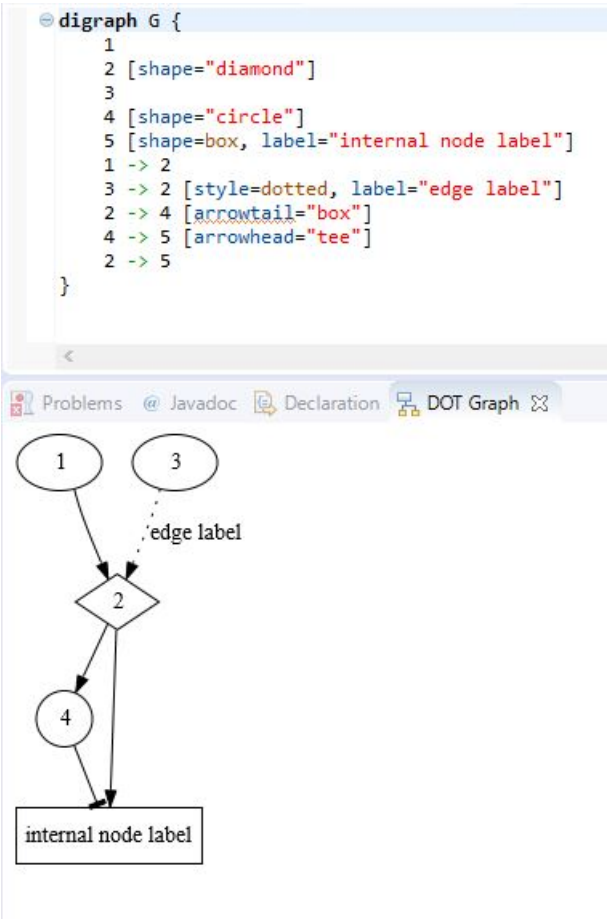


Figura 3.8: Ejemplo de definición de un grafo mediante el lenguaje DOT.

seguido de un nombre. Para definir un nodo simplemente hay que poner un identificador único, seguido (opcionalmente), de unos parámetros para modificar su estilo. Notar que no hace falta separar cada definición por "`;`", pero sí que hay que realizarlas en una línea independiente. Para definir una flecha entre dos nodos, solamente hay que poner el identificador del nodo origen seguido de "`->`" y el identificador del nodo destino. También, se podrán añadir opciones para el estilo de cada flecha. Para una explicación en más detalle sobre el lenguaje DOT, se remite al lector a [Graphviz, 2018a]

Capítulo 4

Estructura de la herramienta

En este capítulo se indaga en la herramienta desarrollada, comentando los dos aspectos más importantes de ésta: la definición de la gramática y la generación de código. Para el primer aspecto se profundizará en la implementación de las reglas más importantes de la gramática (aquellas donde se definen el conjunto de dependencias de las reglas CEP). En la parte de generación de código se hablará cómo se ha conseguido pasar de un AST a un grafo dirigido y cómo se han realizado los análisis y la generación de los diferentes grafos y archivos.

4.1. Entorno

La herramienta desarrollada tiene como objetivo el análisis de las dos propiedades comentada en la sección dedica a los sistemas CEP en el Capítulo 2. A partir de un fichero con la especificación de los eventos y reglas, la herramienta realizará lo que se denomina un análisis estático (análisis sin que el sistema esté en ejecución).

A continuación se explica la arquitectura general de la herramienta y las tecnologías se han utilizado en cada parte, basándonos en la Figura 4.1. El fichero de eventos y reglas será creado dentro de un proyecto en el editor de texto que *Xtext* proporciona, y contendrá una especificación de un programa CEP escrito con el lenguaje de procesamiento de eventos *Esper*. En la Figura 4.1, se muestra que el fichero está fuera del entorno *Xtext* pero, lo hemos representado así ya que el lenguaje de procesamiento de eventos complejos no es parte de Xtext pero cabe aclarar que los programas CEP hay que crearlos en el editor.

Una vez se tenga el fichero de reglas CEP, se podrá analizar las propiedades en la parte de generación de código de *Xtext*, para ello se ha utilizado *Java* y *Xtend* indistintamente para programar los análisis, y el framework *EMF* para acceder al AST que nos proporciona el *parser*.

El primer análisis que se hace es la comprobación de la aciclicidad en las reglas, ya que el resultado que se genera depende de si hay ciclos o no. En el caso de que haya algún ciclo, solo se generará un grafo alertando de dónde se encuentran estos ciclos, es decir, entre qué reglas. Por otro lado, si no hay ciclos se generará un grafo donde se podrán ver las dependencias entre reglas y si las prioridades asignadas antes del análisis son las correctas. Además, se generará el mismo conjunto de reglas entrante pero con una asignación de prioridades que se estiman correctas tras el análisis. Para la visualización de los grafos, en el editor de Eclipse se ha utilizado el framework *GEF* que proporciona una ventana gráfica donde mostrar los grafos escritos en lenguaje *DOT*.

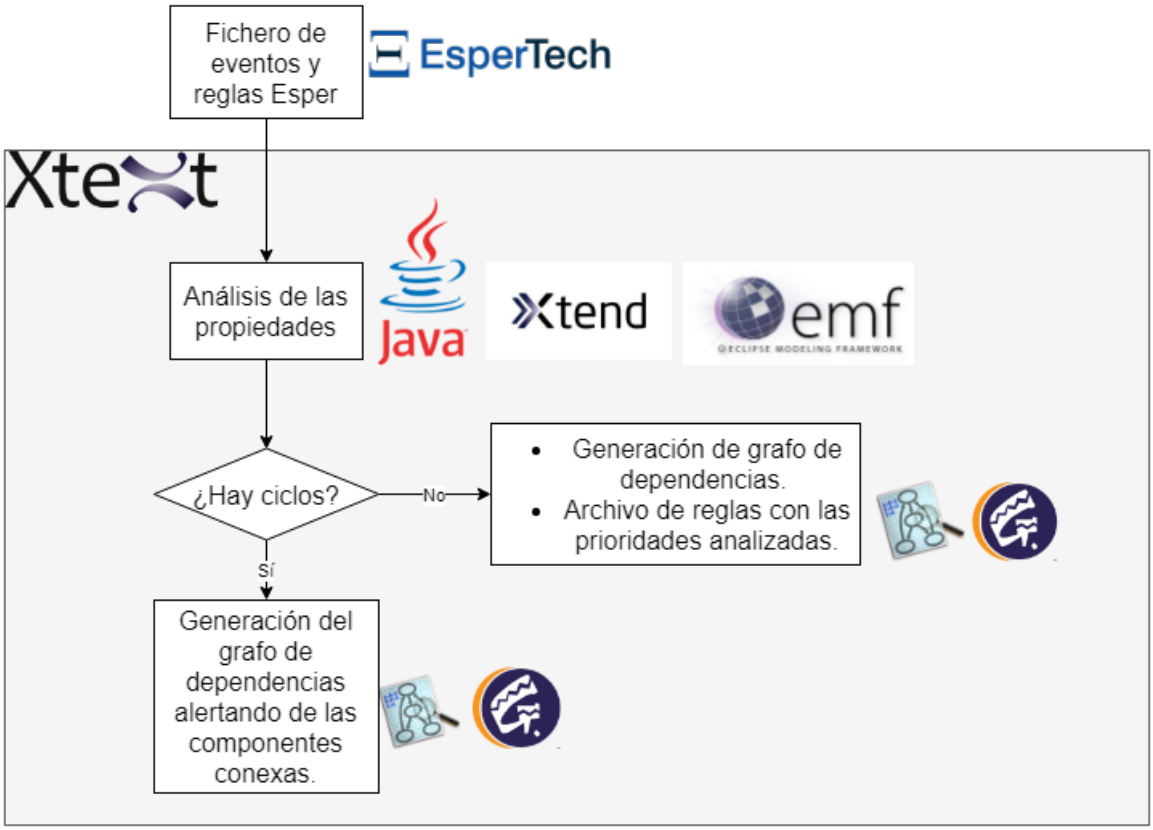


Figura 4.1: *Workflow* de la herramienta y tecnologías utilizadas en cada parte.

Además, independientemente de la existencia de ciclos o no, también se generará un archivo en texto plano que mostrará los resultados obtenidos tras la realización del análisis. Este archivo será una especie de *log* informativo.

4.2. Definición de la gramática

El primer paso para el desarrollo de la herramienta, ha sido el desarrollo de una gramática capaz de reconocer la sintaxis de *Esper*. Para llevar acabo tal fin, se ha utilizado el

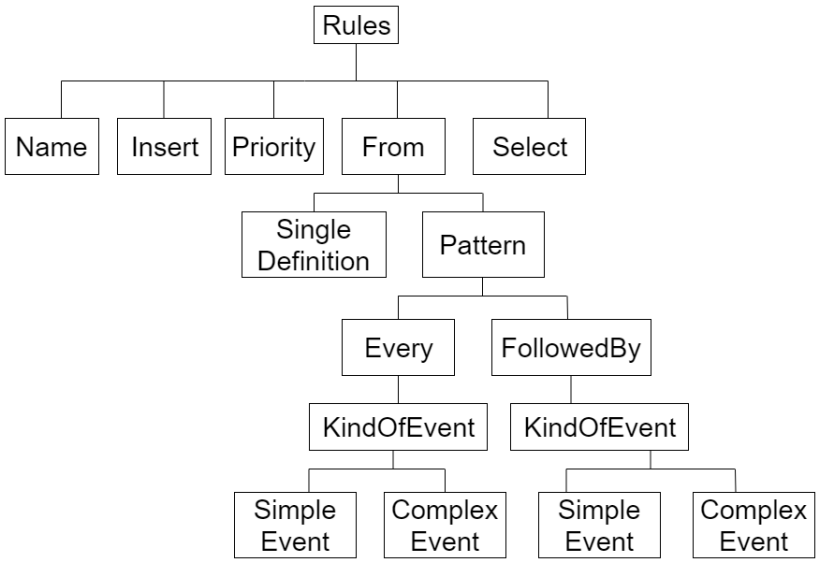


Figura 4.2: Representación de alto nivel del AST generado.

lenguaje de desarrollo de gramática que nos proporciona *Xtext*. Si el lector está interesado, puede ver la definición completa en el Apéndice C.

En la Figura 4.2 podemos observar la estructura (en alto nivel) del AST generado. Se ha obviado la definición de eventos simples ya que ha sido explicado en el Capítulo 2 en la sección dedicada al lenguaje *Esper*. En el desarrollo se ha profundizado con especial énfasis en la parte de la definición de los patrones (sección `from` de las reglas), ya que es aquí donde podemos encontrar las dependencias que tiene una regla y, a raíz de estas dependencias, poder realizar el análisis de las propiedades ya previamente mencionadas. En esta parte, se han utilizado dos operadores principales (`every` y *followedBy*).

A continuación se muestra la definición de la regla *Pattern* (Figura 4.3), la cual representa la sintaxis de los patrones en *Esper*. Ha sido definida con una jerarquía de abstracciones, de tal modo que las regla que expresan un comportamiento de bajo nivel son envueltas por otras reglas que expresan un comportamiento general. Por motivos de claridad, a continuación se explican las reglas sobre su representación gráfica obtenida del visor de reglas de *Xtext* en forma de diagramas (accesible a través de *Window* →*Show View* →*Other...* →*Xtext* →*Xtext Syntax Graph*), las características más importantes de éstas:

- **Pattern:** la Figura 4.4 muestra la representación más abstracta de un patrón, que viene definida por la *keyword* `pattern` y entre corchetes la definición del patrón.
- **JoinFollowBy:** El siguiente paso en la jerarquía de abstracciones viene definido por la regla *JoinFollowBy*, y trata de definir la posibilidad de poder combinar patrones

Figura 4.3: Gramática para la definición de patrones en el lenguaje Esper.

```
Pattern:
        'pattern' '[' joinFollowBy=JoinFollowBy ']' ('.' win=Win)?
;

JoinFollowBy:
        followsByJoinList+=AbstractFollowBy (operator+=Operators followsByJoinList+=
    AbstractFollowBy)*
;

AbstractFollowBy:
        (=> followBy = FollowBy | '(' followBy = FollowBy ')' ) (wherePart=FollowByWhere)?
;


FollowBy:
        leftSide =TerminalExpression (=> '->' rightSide+=TerminalExpression)*
;

TerminalExpression:
        every?='every' everyExpression = FollowBy |
        parenthesis?='(' betweenParenthesis = FollowBy ')' |
        singleDefinition = SingleDefinition
;

KindOfEvent: Event | Insert;

SingleDefinition :
        (=> name=ID '=')? simpleEvents=[KindOfEvent] (=>'('anything=Anything')')?
;
```

mediante operadores, tal y como se puede ver en la Figura 4.5.

- **AbstractFollowBy:** esta regla representa la posibilidad de que cualquier expresión en el patrón esté envuelta (o no) entre paréntesis (Figura 4.6). Nótese que la sección *FollowByWhere* es un complemento añadido para que el *lexer* reconozca el operador `where`, cuya funcionalidad no se explica en este documento ya que no es imprescindible para el funcionamiento de la herramienta. En el caso de que el lector quiera profundizar sobre este operador se le remite a la documentación de Esper [EsperTech, 2018].

- **FollowBy:** con esta regla se representa la abstracción del operador `->`. El operador se define como una expresión terminal seguida de cero o más ocurrencias del operador y otras expresiones terminales, como se puede ver en la Figura 4.7.

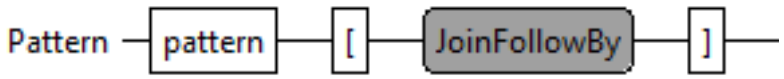


Figura 4.4: Estructura de la regla *Pattern*.

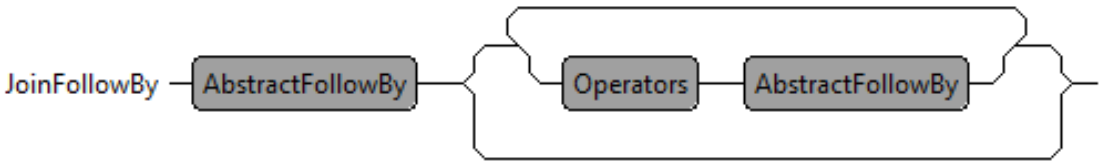


Figura 4.5: Estructura de la regla *JoinFollowBy*.

- **TerminalExpression:** esta regla representa el contenido, tanto de la parte derecha como izquierda, del operador `->`, véase la Figura 4.8. En esta regla nos podemos encontrar tres "subexpresiones":

  1. **SingleDefinition:** definición de un evento simple o complejo mediante su nombre.
  2. **Operador `every`:** operador presentado en el Capítulo 2. Nótese que su definición tiene una componente recursiva al incluir la definición de la regla *FollowBy*.
  3. **Paréntesis:** expresión que hace posible la inclusión del lado izquierdo o derecho entre paréntesis, al depender, de nuevo recursivamente, de la regla *FollowBy*.

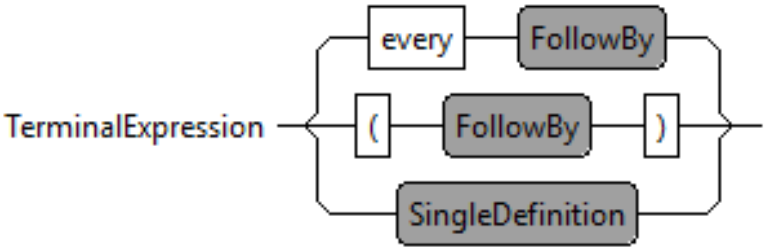


Figura 4.8: Estructura de la regla *TerminalExpression*.

- **SingleDefinition:** con esta regla se hace posible la definición de eventos simples o compuestos en los patrones. En primer lugar tenemos la parte de la asignación que

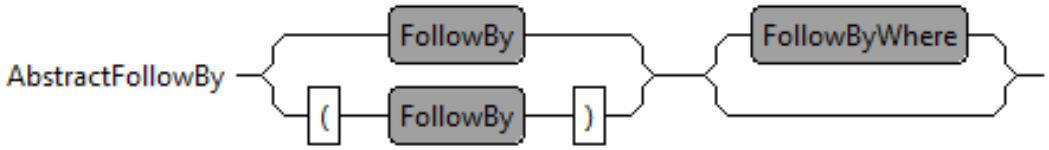


Figura 4.6: Estructura de la regla *AbstractFollowBy*.

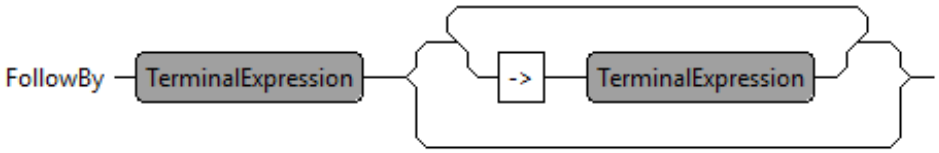


Figura 4.7: Estructura de la regla *FollowBy*.

hace posible asignar un evento a un alias mediante el operador "=". Esta asignación es opcional, y si existe, el alias será el identificador del evento dentro de la regla, con lo cual, en la sección `select`, se podrá referenciar el evento gracias al uso de las referencias cruzadas en *Xtext* (explicado en la sección de *Xtext* del capítulo Capítulo 3). También, haciendo uso de esta característica, se ha podido utilizar los identificadores (previamente definidos) de los eventos y reglas existentes, que en este caso son el proprio nombre del evento simple y el nombre que se le ha dado en la sección `insert into` respectivamente, como un atributo de la propia regla. Esta parte ha sido fundamental para recolectar el conjunto de dependencias de una regla. Esto se ha conseguido mediante el uso de, lo que se denomina, "reglas abstractas". Son reglas que su definición se delegan a otras (del mismo modo que lo hace una clase abstracta en *Java*), del tal modo que, en esta regla se ha realizado una referencia cruzada con la regla "KindOfEvent", la cual delega su implementación en la reglas que define los eventos simples y las reglas, respectivamente.

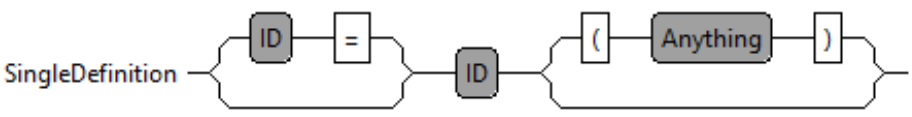


Figura 4.9: Estructura de la regla *SingleDefinition*.

4.2.1. El problema de la gramática ambigua

Una gramática es ambigua cuando para una misma cadena entrante (en este caso sería un archivo con reglas *Esper*) se genera más de un posible árbol de sintaxis abstracta.

El ejemplo más sencillo para representar este problema es en la gramática de una estructura `if-else`. Imaginemos que tenemos las siguientes sentencias:

```
if (condicion1)
   if (condicion2)
     llamadaMetodo1();
   else
     llamadametodo2();
```

Con el código anterior, el *parser* podría generar dos AST, uno donde la sección `else` perteneciera al `if` menos anidado y otro donde perteneciera al que está más anidado. Cuando *Xtext* detecta este tipo de problemas, genera una advertencia cuando se genera la gramática, pero no evita su ejecución ya que selecciona siempre uno de los posibles AST y descarta el resto. No es una buena práctica dejar que *Xtext* haga lo anterior, así que proponen dos posibles soluciones:

1. Deshabilitar el proceso de *backtracking* en el generador ANTLR. Esta opción no se recomienda ya que pueda dar cabida a más problemas a la hora de la generación de la gramática.

2. Utilizar el operador `=>`, el cual hace posible que cuando el *parser* detecte que hay una situación de gramática ambigua, siga explorando por la rama actual en la que se encuentra. Para hacer esto hay que encontrar dónde se está produciendo el problema. Cuando la definición de la gramática es lo suficientemente grande, no es tarea sencilla. La utilización de la herramienta ANTLRWorks (ejemplo de uso en el Capítulo 5) hace posible depurar estos tipos de programas para encontrar este tipo de situaciones. En el desarrollo de la gramática, se ha utilizado esta opción ya que es la recomendada por *Xtext*. Un ejemplo de su uso en este problema planteado, se puede ver a continuación:

```
ReglaIf:
    'if' '(' condicion=ExpresionBooleana ')'
    then=Expresion
    (=>'else' else=Expresion)?
;
```

4.3. Generación de código

Una vez definida la gramática, pasamos a la parte de generación de código. En esta fase se realizan los análisis para determinar la correcta asignación de prioridades y la aciclicidad entre el conjunto de reglas. Además, se genera el grafo resultante del análisis, así como el archivo con una asignación de reglas fruto del análisis (si procede) y un archivo en texto plano que informa al usuario del estado del análisis.

Para poder comprobar las dos propiedades (aciclicidad y orden de ejecución), lo primero que se ha hecho es extraer del AST, que nos proporciona el *parser*, todo el conjunto de dependencias de cada regla y almacenarlo en una estructura de datos que simule un grafo dirigido. Para la extracción del conjunto de dependencias se ha hecho una búsqueda recursiva a través del AST (tal y como se haría en cualquier estructura de datos de tipo árbol), pero ésta ha estado enfocada en las reglas descritas en la sección anterior ya que son todas las que describen el patrón definido.

Al inicio del análisis, se crea un diccionario que *mapea* cada regla con un identificador único (un número entero), ya que varias reglas pueden generar el mismo evento complejo (basta con que el nombre en su sección `insert into` sean los mismos), por lo tanto, si una regla depende de un evento complejo que generan varias reglas, esta regla depende de la ejecución de todas ellas.

Una vez se tiene el diccionario para la conversión entre el nombre del evento complejo y su identificador único, se explora el AST en busca de las dependencias de cada regla. La estructura que emula el grafo dirigido que se quiere conseguir es otro diccionario cuya clave es el identificador único para cada regla, y el valor es una lista con todas las dependencias de ella. De este modo, podemos recorrer el diccionario como si fuera un grafo dirigido, ateniéndonos al conjunto de dependencias de cada clave. Un aspecto a destacar, ha sido la utilización de diccionarios que implementan una función *hash* ("HashMap") y nos proporcionan un acceso, búsqueda y eliminación en tiempo constante ($O(1)$).

A partir del grafo podemos realizar el estudio de la aciclicidad mediante el algoritmo de Kosaraju descrito en la Sección 4.3.1. Este algoritmo nos devuelve una lista con todas las componentes conexas de un grafo ("subgrafos" donde hay ciclos). Si el grafo no tiene ciclos, la lista resultante tendrá tantas componentes conexas como vértices, es decir, estas componentes solo estarán formadas por un único vértice. En el caso de que se detecten ciclos, se generará un grafo en lenguaje DOT mediante el gestor de archivos que obtenemos en la función de *callback*, de tal manera que, para cada componente conexa, sus aristas se mostrarán en color rojo. En la Figura 4.10 se puede ver su implementación, nótese que las

partes sombreadas en color gris son para denotar el espaciado o la escritura de caracteres en la generación de la plantilla.

```
fsa.generateFile(fileName + '.dot',
    '''
    digraph «fileName» {
        «FOR entry : mapSimpleEventToString.entrySet»
        «entry.value» [shape="box", label="«entry.key»"];
        «ENDFOR»
        «FOR entry : mapRulePartsToString.entrySet»
        «entry.value» [shape="oval", label="«entry.key.name»"];
        «ENDFOR»
        «FOR entry : mapUniqueId.entrySet»
            «FOR dependencie : entry.value»
            «IF isThereAnySetContainingThisPairOfNodes(entry.key, dependencie, scc)»
                «dependencie» -> «entry.key» [color=firebrick1]
            «ELSE»
                «dependencie» -> «entry.key»
            «ENDIF»
            «ENDFOR»
        «ENDFOR»
    }
    '''
)
```

Figura 4.10: Implementación de la generación del archivo `.dot` donde se visualizará el grafo.

Si, tras el análisis de aciclicidad, no se detectarán ciclos, se genera un grafo de dependencias, con la peculiaridad de que para cada regla se va a generar una prioridad según su orden topológico (se puede ver una explicación de este algoritmo en la Sección 4.3.1). A la hora de generar este grafo, si la prioridad que se le ha sido asignado a cada regla no coincide con la prioridad según su orden topológico, el vértice se mostrará en color rojo. Esto no quiere decir que la asignación de la prioridad sea errónea, si no que no coincide con esta asignación de prioridades.

Para realizar esta asignación de prioridades se ha adaptado el algoritmo del orden topológico a una forma recursiva. Es decir, para cada vértice, su prioridad será la máxima prioridad de todos sus vértices predecesores, sumándole una unidad más. El caso base son los eventos simples cuya prioridad se le ha asignado el valor -1 (en *Esper* no se pueden asignar prioridades a los eventos simples), de tal forma que, si una regla depende únicamente de un evento complejo, la prioridad de ésta será $-1 + 1 = 0$ (máxima prioridad posible).

Tras la generación del grafo, y ya teniendo las prioridades asignadas, se genera el mismo archivo con las reglas CEP de entrada al cual se le han asignado prioridades según su orden topológico. Si alguna de las reglas de entrada tenía alguna prioriad incorrecta, es decir, su prioridad no corresponde con la prioridad asignada mediante el análisis, se

genera el archivo de reglas pero en las secciones `@Priority`, estará definida la prioridad según en análisis.

De antemano, se puede pensar que le implementación de esta funcionalidad ha sido directa, ya que solo bastaría con recorrer el AST e ir iterando sobre todas las reglas cambiando la sección `@Priority`. El problema radica en que la generación de código se realiza después de que el *parser* genere el AST. Este, a su vez, es construido con los *tokens* que realiza el *lexer* sobre el archivo. En este proceso de *tokenización* se eliminan todas las *keywords* definidas en las reglas y se genera los atributos de cada regla con el valor que les corresponda. De este modo, con el AST se pierde el acceso a las *keywords* definidas. Para solventar el problema, se ha tenido que indagar en la API de *EMF* para obtener un método que devuelva una URI (*Uniform Resource Identifier*) del AST generado (está *URI* representaría la dirección del archivo de reglas, del cual se ha generado el AST). Esta URI obtenida es una implementación de la propia API de EMF y no una estandarizada, con lo cual se ha tenido que utilizar un método de esta API que crea un *InputStream* a partir de este tipo de *URIs*. En la Figura 4.11 se pueden ver los métodos utilizados para llevar este proceso a cabo. Una vez obtenido el acceso al archivo, simplemente se ha iterado sobre cada línea de cada regla, modificando la sección `@Priority` por la prioridad que le corresponde del análisis.

```
ResourceSet rs = new ResourceSetImpl();
URIConverter uriConverter = rs.getURIConverter();
StringBuilder sb = new StringBuilder();

InputStream inputStream = null;
try {
    inputStream = uriConverter.createInputStream(fileLocation);
} catch (IOException e1) {
    e1.printStackTrace();
}
```

Figura 4.11: Apertura de un *InputStream* a partir de la representación de URI de EMF.

4.3.1. Algoritmos utilizados

Orden topológico de un grafo

Dado un grafo dirigido acíclico (no contiene ningún ciclo entre sus vértices), un orden topológico es una relación de orden total ($\prec$) entre vértices tal que: si existe un arco desde n a m, entonces m es mayor que n en el orden. Para un mismo grafo dirigido puede haber diferentes ordenes topológicos, y si el grafo dirigido contienen algún ciclo, este orden no se podrá calcular. Normalmente este algoritmo es usado para tareas de planificación de

tareas con dependencias entre ellas (es decir, una tarea no se puede completar si la ejecución de otra u otras).

Para explicar cómo realizar una ordenación topológica sobre un grafo dirigido, se introducen dos conceptos:

- **Fuente**: vértice que no recibe ninguna arista (su grado de entrada es 0).
- **Sumidero**: vértice del cual no salen ningún vértice (su grado de salido es 0).

Tomemos de partida el grafo inicial de la Figura 4.12, cuyos vértices representan tareas a realizar y sus aristas dependencias entre ellas. De este modo, la tarea 3 solo se podrá hacer si previamente se ha hecho la tarea 1 y la tarea 2. Para calcular un orden topológico sobre este grafo basta con seleccionar una fuente y visitarla quitando todas sus aristas. Este proceso se repite hasta que se hayan visitado todos los vértices. El orden en el cual se han ido visitando los vértices, sería un posible orden topológico.

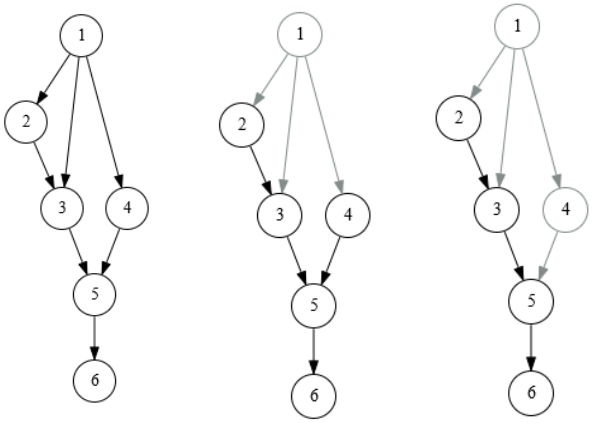
Siguiendo el ejemplo, empezaríamos en la única fuente del grafo, la tarea 1. Visitamos el vértice $\{1\}$ y eliminamos todas sus aristas. A continuación nos quedarían dos posibles fuentes, la tarea 2 y la tarea 4. Seleccionamos la tarea 4 y repetimos el proceso, quedando el conjunto de vértices seleccionados: $\{1, 4\}$. Si repetimos el proceso anterior, una vez visitados todos los nodos, obtendremos el siguiente orden topológico: $\{1, 4, 2, 3, 5, 6\}$.

Algoritmo de Kosaraju

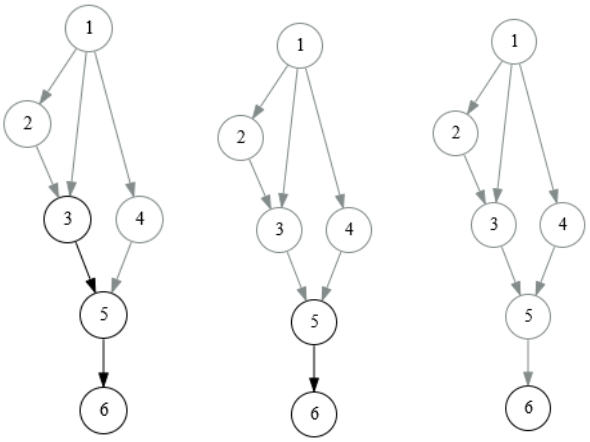
Dado un grafo dirigido, una componente fuertemente conexa es un subconjunto de los vértices que componen el grafo, donde hay un camino entre dos vértices cuales quiera y otro camino de vuelta, es decir, hay un ciclo en ese conjunto de vértices.

El algoritmo de Kosaraju es un algoritmo que calcula todas las componentes conexas en un grafo dirigido. Su complejidad es linear ya que el algoritmo crece en proporción al numero de vértices y aristas en el grafo, es decir, su complejidad es $O(V + E)$. Para la ejecución de este algoritmo necesitaremos un conjunto para marcar los vértices visitados y una estructura de datos *LIFO* (por sus sigas en inglés *Last In, First Out*), con una simple pila es suficiente. Los pasos a seguir serían los siguientes:

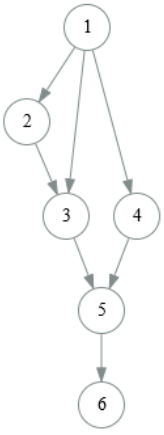
1. Comenzamos visitando un vértice, desde este se realiza una búsqueda en profundidad (*DFS*) visitando todos los vértices en su recorrido. Cuando un vértice no se pueda expandir más (desde él no se pueda navegar hacía otro vértice) será añadido a nuestra pila. Una vez que el vértice donde hemos empezado la búsqueda en profundidad no tiene más vértices que no hayan sido visitados, se añadirá a la pila y seleccionaremos otro vértice no visitado para repetir este proceso.



Estado Inicial Paso 1 Paso 2



Paso 3 Paso 4 Paso 5



Paso 6

Figura 4.12: Elaboración de un orden topológico.

2. Una vez que todos los vértices hayan sido visitados, tenemos que invertir el grafo, es decir, construir el mismo grafo pero invirtiendo la dirección de sus aristas. Si tenemos una aristas de A hacia B, al invertirla, esta aristas irá de B hacia A.

3. Ahora, partiendo de un conjunto vacío de vértices visitados, se realiza una búsqueda en profundidad de los vértices que están en la pila. De esta manera, se empieza extrayendo un vértice de la pila, comprobando si no ha sido previamente visitado (si ha sido visitado, se descarta y se extrae el siguiente) y se realiza una búsqueda en profundidad sobre el grafo inverso.

4. En la búsqueda en profundad se irán visitando todos los vértices hasta que no se puedan visitar más (porque ya hayan sido visitados). El conjunto de vértices visitados para esta búsqueda concreta sería una componente conexa. Para encontrar el resto de componentes, bastaría repetir el paso 3 y 4 hasta que nuestra pila se quede vacía.

Tomando como ejemplo el grafo situado en la parte izquierda de la Figura 4.13, un conjunto vacío de vértices visitados C_1 y una pila vacia P, vamos a aplicar este algoritmo. Empezando por el vértice B, aunque podríamos empezar por cualquier otro, realizamos una búsqueda en profundidad hasta llegar al vértice D, desde el cual no podemos seguir navegando ya que no tienes más sucesores. Al llegar a este nodo, el conjunto de visitados C_1 incluiría todos los vértices del grafo $\{\text{B, C, A, D}\}$ e incluiríamos el nodo D en la pila $P = \{D\}$. A continuación comenzaría el preceso de *backtracking*, es decir volver al vértice desde el cual hemos llegado al vértice D y comprobar si tienes más sucesores no visitados para explorar. Una vez realizado el proceso completo, nuestra pila quedaría de la siguiente manera, $P = \{B, C, A, D\}$.

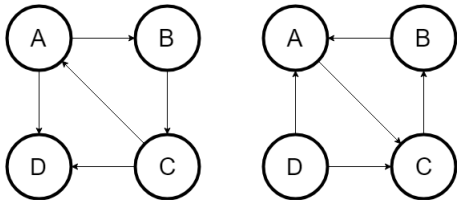


Figura 4.13: A la izquierda un grafo dirigido. A la derecha el grafo anterior invertido.

Una vez tengamos la pila con el orden de visita de los vértices, tenemos que invertir el grafo (grafo situado en la parte derecha de la Figura 4.13) y aplicar, de nuevo, una búsqueda en profundidad sobre los vértices extraídos de la pila. De esta manera, el primer vértice extraído sería el vértice B y su búsqueda en profundidad daría como resultado la componente conexa $\langle B, A, C\rangle$. El siguiente vértice que se extraería de la pila sería el vértice D ya que los vértices C y A ya han sido visitados en la búsqueda en profundidad

anterior. Al hacer la búsqueda en profundidad sobre el vértice D sus nodos adyacente ya han sido visitados con lo cual nos queda otra componente conexa formada por el vértice D, $\langle D \rangle$. Al extraer el último elemento de la pila, está quedaría vacía, con lo cual, el algoritmo finalizaría y como resultado obtendríamos las dos componentes conexas anteriores.

Capítulo 5

Validación y pruebas

Este capítulo detalla la depuración de la gramática desarrollada, y cómo se ha delimitado el desarrollo de ésta. Para realizar la depuración, se introduce el uso de la herramienta *ANTLRWorks* para depurar gramáticas ANTLR, este tipo de gramáticas son las que utiliza *Xtext* internamente. Además, se propone un conjunto de reglas que han servido como test de regresión.

5.1. Depuración de la gramática

En el desarrollo de la gramática, el problema más frecuente que ha surgido, es que en algún punto de su desarrollo ésta se volvía ambigua. Cuando la gramática desarrollada es pequeña, detectar este tipo de problemas puede ser relativamente sencillo a partir de las alertas que genera *Xtext*. Pero cuando la gramática se vuelve más extensa e incluye reglas recursivas que depende de otro tipo se reglas, detectar este problema se hace más difícil.

Xtext no proporciona ningún método para depurar la gramática. Si hay errores en la definición, el *workflow* que genera tanto el *lexer* como el *parser*, no se ejecuta y se lanza un error. Si *Xtext* detecta que la gramática es ambigua, pero su definición es correcta, el *workflow* se ejecuta pero, internamente *Xtext* poda el AST cuando detecta ambigüedad. No es muy recomendable dejar a *Xtext* realizar este tipo de acciones porque se pierde el control de la gramática y podemos estar generando un AST que no es acorde a nuestra definición.

Por este motivo se ha utilizado el entorno de desarrollo para gramáticas ANTLR *ANTLRWorks* [ANTLR, 2018]. Esta herramienta nos permite visualizar de una forma gráfica aquellas reglas que tienen problemas de ambigüedad, mostrando un diagrama donde se pueden ver distintas trazas de ejecución que generan el mismo AST. La herramienta se descarga como un fichero `.jar`, de tal modo que su ejecución es directa si se tiene instalado una *JVM* en la máquina pertinente.

Lo primero que hay que hacer para utilizar esta herramienta, es configurar el *workflow* en *Xtext* para que genere un archivo con extensión `.g`, el cual contendrá la gramática ANTLR. No se puede utilizar la gramática ANTLR generada directamente por *Xtext* ya que éste, en última instancia, la modifica para incluir aspecto de autocompleción o verificación del código. En la Figura 5.1 se puede ver el contenido del *workflow* incluyendo la sentencia correspondiente para activar la generación de la gramática ANTLR a depurar (líneas 43-45).

```
8  Workflow {
9
10     component = XtextGenerator {
11         configuration = {
12             project = StandardProjectConfig {
13                 baseName = "org.xtext.example.PruebaCompConexas"
14                 rootPath = rootPath
15                 runtimeTest = {
16                     enabled = true
17                 }
18                 eclipsePlugin = {
19                     enabled = true
20                 }
21                 eclipsePluginTest = {
22                     enabled = true
23                 }
24                 createEclipseMetaData = true
25             }
26             code = {
27                 encoding = "windows-1252"
28                 lineDelimiter = "\r\n"
29                 fileHeader = "/*\n * generated by Xtext \${version}\n */"
30             }
31         }
32         language = StandardLanguage {
33             name = "org.xtext.example.mydsl.MyDsl"
34             fileExtensions = "mydsl"
35
36             serializer = {
37                 generateStub = false
38             }
39             validator = {
40                 // composedCheck = "org.eclipse.xtext.validation.NamesAreUniqueValidator"
41             }
42
43             parserGenerator = {
44                 debugGrammar = true
45             }
46
47         }
48     }
49 }
50
```

Figura 5.1: *Workflow* para la generación de la gramática.

Una vez configurado el *workflow*, generamos la gramática como se ha explicado en la sección dedicada a *Xtext* del Capítulo 3. Cuando se haya generado, dentro del directorio "src-gen" del proyecto principal, en el paquete cuya terminación es "...antlr.internal", encontraremos el archivo con extensión `.g` que contienen una gramática apta para ejecutarla en *ANTLRWorks*.

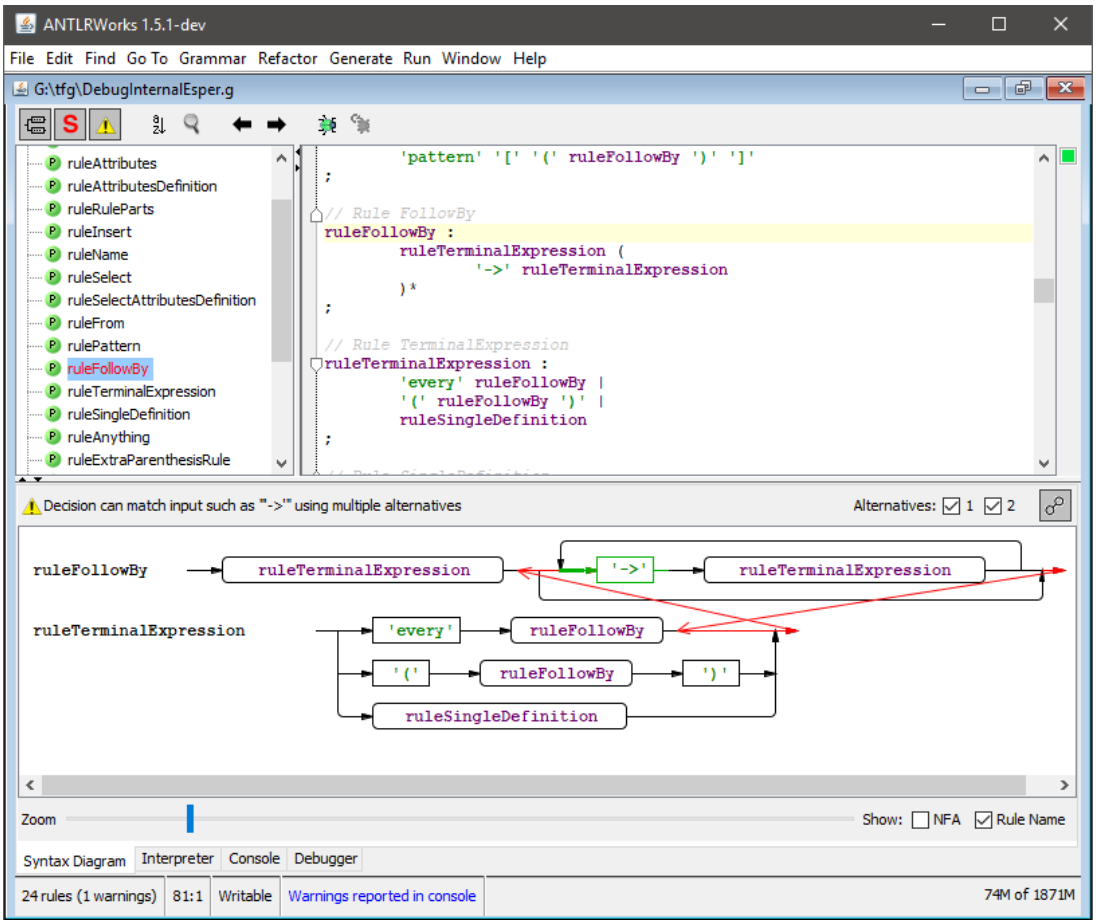


Figura 5.2: Ejemplo de depuración de gramática ANTLR en *ANTLRWorks*.

A continuación, abrimos *ANTLRWorks* y cargamos el fichero `.g` (*File* → *Open*). Se nos desplegará en la parte izquierda un listado de todas las reglas contenidas. Si pulsamos sobre el botón para hacer la depuración (botón con icono de un insecto), se efectuará un análisis sobre todas las reglas y, al finalizar el análisis, aparecerán en rojo aquellas reglas que contengan algún tipo de problema. Si pulsamos sobre alguna de ellas, se nos abrirá un diagrama con la estructura de la regla. Sobre este diagrama podemos superponer las diferentes trazas que generan el mismo AST, pulsando en la sección "Alternatives" y seleccionando la traza o trazas que deseemos mostrar. En la Figura 5.2, se puede ver dos trazas superpuestas (verde y roja) en una regla que produce un problema de ambigüedad en la gramática. Gracias a esta herramienta se ha podido depurar la gramática al darnos los puntos exactos donde se genera la ambigüedad, para corregirlos se ha hecho uso del operador `=>` explicado en el Capítulo 4.

5.2. Pruebas de regresión

Las pruebas de regresión sirven para comprobar que una nueva funcionalidad añadida a un software, no modifica o altera una o varias funcionalidades previamente existentes

en el programa. Estas pruebas se van construyendo a raíz de los casos de prueba para comprobar el correcto funcionamiento de una funcionalidad, de tal manera que cuando se desarrolle una nueva característica, contaremos con una batería de pruebas que cubrirá cada funcionalidad anterior a la nueva desarrollada. Si esta nueva funcionalidad pasa estas pruebas de regresión, se podrá afirmar, en mayor o menor medida, que no modifica las demás funcionalidades preestablecidas.

Un punto de especial interés en el desarrollo del presente proyecto, ha sido decidir el nivel de detalle a incluir en nuestra gramática para reconocer programas CEP. Desarrollar desde cero un *lexer* capaz de reconocer el lenguaje completo Esper, no tendría mucho sentido, ya que podríamos utilizar el propio *lexer* que implemente Esper y ampliarlo. Además siguiendo esta vía de usar software desarrollado por terceros, se perdería el componente académico del aprendizaje de todas las tecnologías principales utilizadas mediante su desarrollo desde cero.

Para delimitar el alcance del *lexer*, durante el desarrollo del mismo se han definido cuatro conjuntos de reglas que han servido de pruebas de regresión y de delimitadores. Debido a que en el desarrollo de la gramática hay reglas con referencias cruzadas hacia otras, jerarquías de reglas y recursividad, es muy probable que al intentar modificar alguna o intentar añadir una funcionalidad sobre reglas existentes, se acaben alterando el funcionamiento de las reglas de la gramática. Con tal motivo, para cada ampliación del *lexer*, se hace comprobar, a partir de estos conjuntos de reglas considerados como un conjunto de pruebas de regresión, que no se han alterado los patrones ni estructuras que ya se reconocían previamente.

Por otro lado, han servido como delimitadores en el sentido de que cuando el *lexer* ha sido capaz de cubrir los 4 conjuntos de reglas, se ha dado por finalizado su desarrollo para poder pasar a la fase de análisis y generación de código. Cada conjunto de reglas tiene un aspecto peculiar que lo diferencia del resto, de este modo, para una versión del *lexer* que reconociera un conjunto determinado, no reconocería los restantes.

A continuación se muestra una breve descripción de estos conjuntos de reglas:

- *Smart House* [Moreno et al., 2018]: este conjunto consta de seis reglas que modelan un sistema CEP en una "hogar inteligente". Ese conjunto ha sido el pilar fundamental para el desarrollo de la estructura `pattern`.
- *Motorbike* [Burgueño et al., 2018]: en este conjunto encontramos un sistema CEP que se ejecuta sobre diferentes sensores en una motocicleta. De este conjunto de reglas se extrajo la capacidad de que el *lexer* fuera capaz de reconocer "joins" (patrones

de patrones).

- *Air Quality* [Burgueño et al., 2018]: conjunto de más de cuarenta reglas que modelan un sistema CEP para la detección de los diferentes componentes que hay en el aire. La característica más destacable es que la mayoría de reglas producen el mismo tipo de evento que recolecta una única regla, con lo cual, ésta dependía de todas las anteriores.

- *Nuclear power station* [Boubeta-Puig, 2018]: este conjunto de reglas modela un simple sistema de alerta en una central nuclear cuando se detecta cierto patrón de temperatura. El conjunto ha sido tomado como punto de partida para el desarrollo de la gramática al contar con pocas reglas pero conteniendo la estructura básica de éstas.

En el Apéndice D se puede se encuentran estos conjuntos de reglas Esper..

Capítulo 6

Conclusiones

En este capítulo se exponen las conclusiones fruto del desarrollo de esta herramienta y de los conocimientos obtenidos para la consecución de la misma, así como, las líneas futuras de ampliación de este proyecto.

6.1. Desarrollo de la herramienta

El desarrollo de la herramienta ha sido la parte más costosa del proyecto, ya que se ha tenido que aprender desde cero todas las tecnologías utilizando (excluyendo el lenguaje de programación *Java*). Para llevar a cabo los análisis sobre los programas CEP escritos en Esper, se ha elaborado de un DSL cuya sintaxis coincide con la sintaxis de lenguaje de procesamiento de eventos Esper. Una de las partes más complejas de la herramienta ha sido la definición de la gramática, y para su desarrollo se ha utilizado una herramienta específicamente diseñada para depurar programas basados en gramáticas.

Una vez que se ha definido la gramática, se ha podido reconocer conjuntos de reglas Esper y proceder a realizar los análisis en su especificación. Para llevar a cabo estos análisis se ha tenido que modelar los conjuntos de reglas Esper como un grafo dirigido de dependencias y, sobre éste, aplicar algoritmos que analicen sus componentes conexas y el orden topológico.

La idea principal para mostrar los resultados de los análisis, es generar una ventana, dentro del propio entorno de desarrollo Eclipse, donde se visualice el grafo resultante. Se decidió utilizar el *framework* GEF que nos permite generar un interprete del lenguaje DOT, lenguaje diseñado para la representación de grafos, de forma nativa en Eclipse. De este modo, la generación de los grafos se hizo con este lenguaje.

6.2. Líneas futuras de ampliación

Una vez realizado el proyecto, se proponen las siguientes líneas de ampliación que aprovechan todo lo desarrollado en este TFG.

En primer lugar, utilizando la gramática desarrollada, se propone la ampliación de ésta para que reconozca más aspecto del lenguaje Esper tales como:

- Operadores adiciones: en Esper existen otros operadores que no han sido utilizados en este TFG. Hacer que la gramática pueda reconocerlos aumentaría la extensibilidad para la realización de otro tipos de análisis basado en la semántica de estos nuevos operadores.
- Funciones predefinidas: Esper cuenta con un gran repertorio de funciones predefinidas (suma, resta, cálculo de medias, etc.). En el *lexer* desarrollado solo se tienen en cuanta las funciones que encontramos en el conjunto de reglas de validación. Enriquecer la gramática con más funciones predefinidas supone que el *lexer* reconozca más patrones.
- Reconocimiento de ventanas de tiempo: una características de Esper muy interesante, es la posibilidad de definir ventanas de tiempo. Haciendo que la gramática pudiera reconocerlas, se podrían generar nuevos análisis basados en estas ventanas temporales.

Por otro lado, en la mayoría de los casos, los programas CEP implican su ejecución en entornos de tiempo real. En estos tipos de entornos es muy importante contar con mecanismos que calculen la incertidumbre (probabilidad de ocurrencia) de eventos simple y complejos. Siguiendo el enfoque presentado en [Moreno et al., 2018], se podría extender este TFG para que dado un programa CEP sin incertidumbre, se generara el código Esper correspondiente teniendo en cuenta dicha incertidumbre y su propagación.

Por último, si un usuario quiere utilizar la herramienta debería instalar todas las dependencias en su máquina e importar el proyecto en Eclipse. Lo ideal sería generar un *plug in* para Eclipse donde se instalaran todas las dependencias automáticamente y se reconociera el lenguaje sin tener que abrir el editor de texto en segunda instancia.

Apéndice A

Manual de instalación

En este apéndice se mostrará la instalación de todas las herramientas de las que depende la ejecución del proyecto. Nótese que el desarrollo del proyecto se ha realizado sobre el sistema operativo *Windows*, y el proceso de instalación podría variar para otros sistemas operativos.

A.1. Eclipse

En este proyecto se ha utilizado la versión *Oxigen* del entorno de desarrollo Eclipse, la cual es la última versión lanzada hasta la fecha. Para descargarlo, basta con acceder a la siguiente dirección web, https://www.eclipse.org/downloads/eclipse-packages/, donde encontraremos un listado de las diferentes versiones de este IDE, para este caso se ha utilizado la versión "*Eclipse IDE for Eclipse Committers*". A continuación, deberemos seleccionar la versión a descargar, 32 o 64 bits, y la descarga comenzará automáticamente.

Una vez descargado, basta con descomprimir el archivo y obtendremos una carpeta con el nombre *eclipse*, que accediendo a ella, encontraremos el archivo de ejecución para lanzar el IDE, cuyo nombre es el mismo que el de la carpeta pero con la extensión `.exe`.

A.2. Xtext

Una vez instalado el entorno de desarrollo Eclipse, podemos instalar el *plug-in Xtext* siguindo estos pasos:

1. Seleccionamos *Help* →*Install New Software...*. A continuación se nos abrirá una ventana y pulsaremos sobre el bonton *Add...*

2. Introducimos el siguiente enlace en la sección *Location*: http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/. Aceptamos y se nos abrirá el listados de herramienta del repositorio añadido.

3. En listado de las diferentes herramientas, seleccionamos *Xtext* →*Xtext Complete SDK* y aceptamos todas las ventanas hasta llegar a la última que nos pedirá confirmación para la instalación.

4. Una vez instalado, Eclipse solicitará reiniciar la aplicación para poder efectuar los cambios realizados. Después del reinicio, se tendrá completamente instalado *Xtext* y se podrá crear proyectos como se ha visto en la sección destinada a *Xtext* en el Capítulo 3.

A.3. GEF (Graphical Editing Framework)

Para la instalación de este framework, se deben seguir estos pasos:

1. Acceder al siguiente enlace: `http://www.eclipse.org/downloads/download.php?file=/tools/gef/downloads/drops//5.0.1/R201708280201/GEF-Update-5.0.1.zip`, en él podremos descargar este framework en forma de archivo comprimido. Nótese que se ha utilizado la versión 5.0.1 del framework, pero el proceso de instalación es indiferente a la versión del mismo.

2. En eclipse, seleccionamos *Help* →*Install New Software...* y, a continuación, pulsamos en el botón "*Add...*".

3. En la venta que aparece tras pulsar el botón anterior, seleccionamos el botón *Archive...* y seleccionamos el archivo comprimido a través de su ruta en el sistema.

4. Una vez especificada la ruta donde se encuentra el archivo comprimido, pulsamos sobre el botón *OK*, y se muestra todos los componentes que se pueden instalar de forma aislada a partir de este framework. En nuestro caso, seleccionamos todos los paquetes pulsando sobre el botón *Select All*.

5. Pulsamos en *Next* en todas las ventanas hasta que se produzca la instalación. En el momento que se realice, Eclipse pedirá que se reinicie para aplicar todos los cambios.

Una vez instalado, para abrir el interprete del lenguaje DOT, el cual nos proporciona una representación gráfica de grafos escritos en lenguaje DOT, tendremos que seleccionar *Window* →*Show view* →*Other...* →*Visualization* →*DOT Graph*. Para enlazar un archivo `.dot` al interprete abierto, solo se tendrá que pulsar sobre el botón "toggle sync mode" que hay abajo de la vista

A.4. Graphviz

La instalación de este motor para la visualización de grafos es opcional pero muy recomendable ya que hace posible una visualización más optimizada dentro del entorno de desarrollo Eclipse. Sus pasos par la instalación son los siguientes:

1. Acceder al siguiente enlace y descargar el archivo con extensión `.msi` (https://graphviz.gitlab.io/_pages/Download/Download_windows.html)
2. Una vez descargado, instalarlo siguiendo el asistente de instalación.

Una vez instalado, hay que configurar eclipse para que sepa donde se ha instalado este *plug-in*. Para ello hay que pulsar sobre *Window* →*Preferences* →*DOT* →*Graphviz* y en la sección "Location", poner la ruta donde se ha instalado el archivo `dot.exe`. Por defecto la ruta de instalación sería la siguiente: `C:\Program Files (x86)\Graphviz2.38\bin\dot.exe`. En la parte inferior de esta venta se puede encontrar los formatos para exportar el grafo generado, por defecto su exportación ser realizará en `pdf`.

Apéndice B

Manual de usuario

En este apéndice se mostrara un ejemplo de uso completo de la herramienta. Para ello se partirá del conjunto de reglas *Smart House* y del editor de texto ya desplegado en segunda instancia (ambos explicados en el Capítulo 5 y en el Capítulo 3 respectivamente). Nótese que la extensión utilizada para los archivos de reglas ha sido `.esper`.

Una vez abierto el editor de texto y creado un proyecto para albergar el archivo de reglas, simplemente tendremos que guardar el archivo para que se realice el análisis y se genere los archivos pertinentes. Como se observa en la Figura B.1 se han generado tres archivos en el directorio `src-gen`:

- `SmartHouse.dot`, el cual representa el grafo que será interpretado por el visor.
- `SmartHouseLog.txt`: *log* del análisis. Se puede ver su estructura en la Figura B.1.
- `SmartHousePriorities.esper`: archivo con las mismas reglas entrantes pero con las asignación de prioridades acorde a un orden topológico de las reglas.

Como se puede observar en la Figura B.1, tras el análisis no se han detectado ciclos pero se han detectado reglas cuyas prioridades no están acorde con el orden topológico generado. Si abrimos el archivo `SmartHouse.dot` y enlazamos este archivo con la vista que nos proporciona GEF (explicado en la sección sobre GEF del Apéndice A) obtendremos la representación gráfica del grafo, tal y como se puede ver en la Figura B.2. Como se observa en esta imagen, los eventos simples se han representado con una forma rectangular y los eventos complejos con una forma de óvalo. Además, hay dos vértices cuya prioridad no se corresponde con la del análisis (vértices en rojo). Abriendo el archivo `SmartHousePriorities.esper` nos encontrarnos el conjunto de reglas y sus prioridades acorde a este análisis.

Partiendo del mismo conjunto de reglas vamos a provocar un ciclo sobre su especificación para mostrar como difiere la generación de los archivos según se detecten ciclos o no.

Una vez insertado el ciclo en la especificación, volvemos a guardar el archivo y la herramienta nos generará dos archivos (*log* y la representación del grafo). Como podemos observar en la Figura B.3, en el archivo `SmartHouseLog.txt`, se nos muestra que se ha detectado ciclos en la especificación y, además, cuáles son las componentes conexas de cada ciclo. Como ya se ha comentado anteriormente, si se detectan ciclos, no es posible generar las prioridades de cada regla porque, para hacerlo, se necesita que no existan ciclos.

Observando la Figura B.4, podemos ver el grafo que se genera pero, esta vez, se muestra aristas en color rojo, que son las componentes conexas del ciclo detectado.

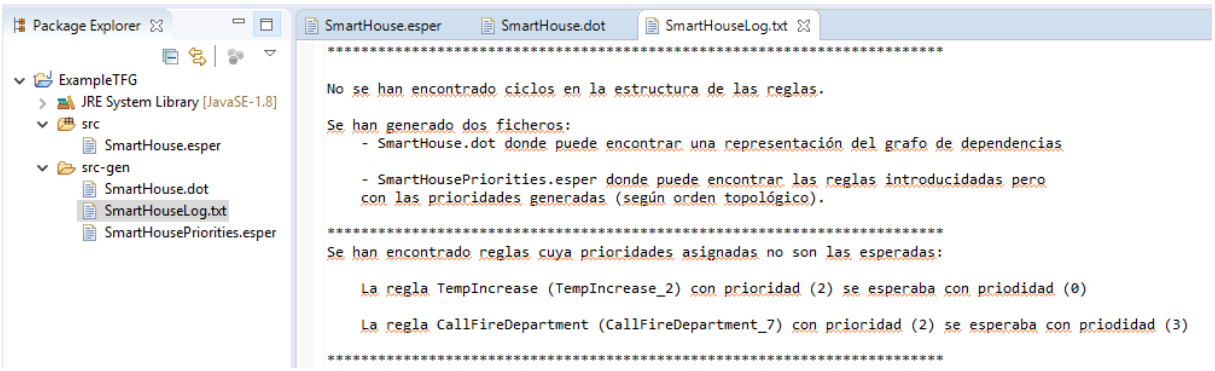


Figura B.1: *Log* generado tras el análisis.

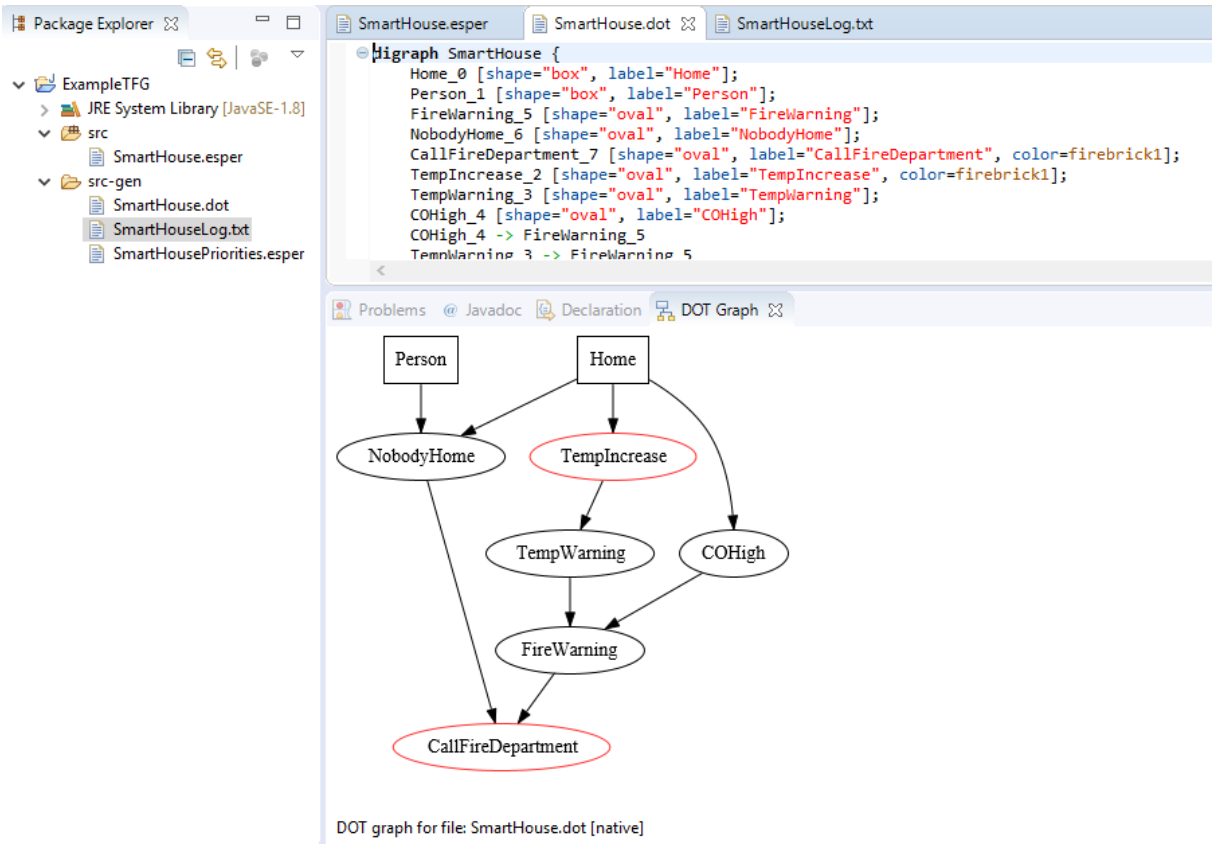


Figura B.2: Grafo generado tras en análisis.

Figura B.3: *Log* generado tras el análisis insertando un ciclo.

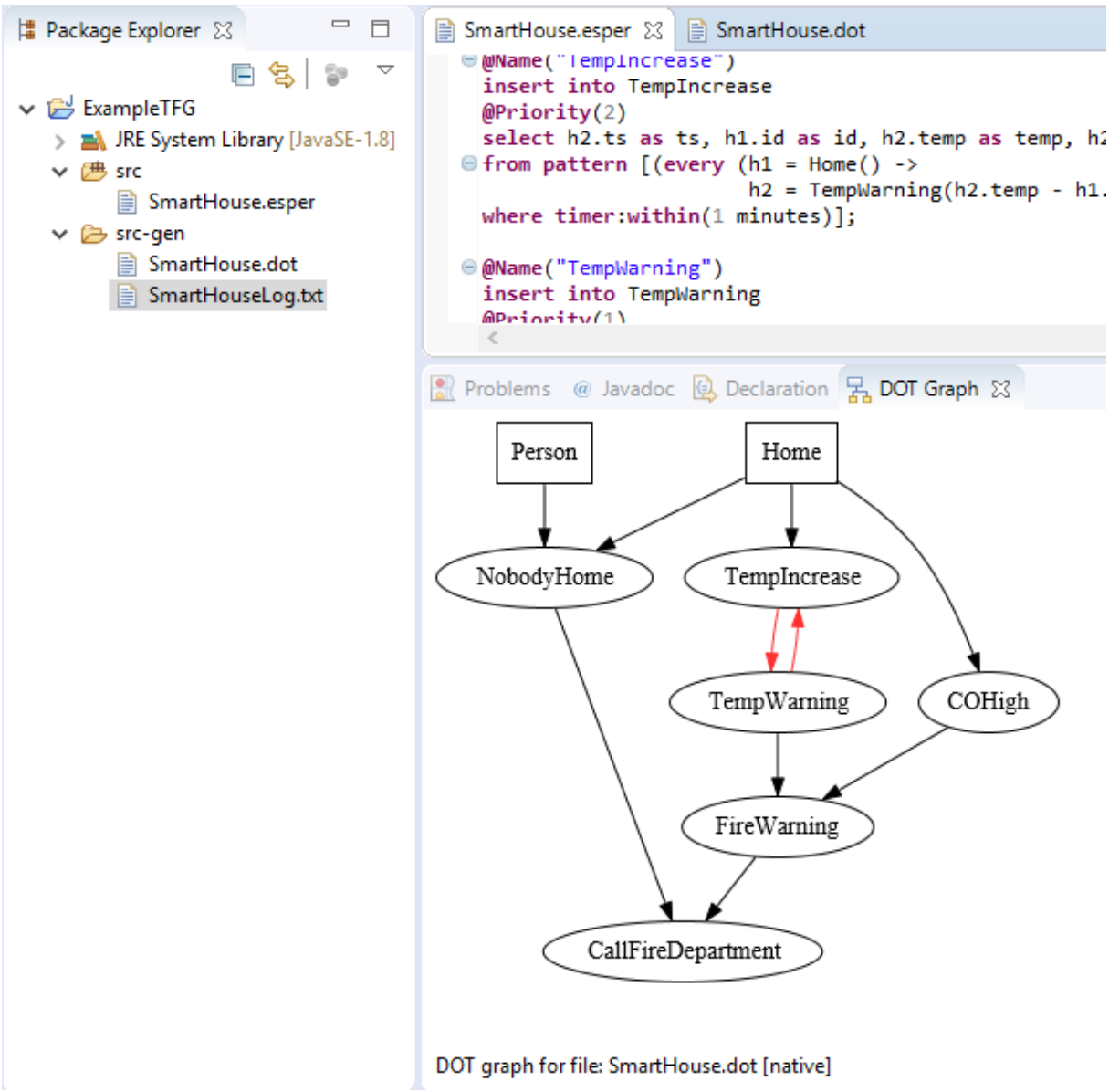


Figura B.4: Grafo generado tras el análisis insertando un ciclo.

Apéndice C

Definición de la gramática utilizada en el lexer

En este apéndice se muestra la definición completa de la gramática del lenguaje de procesamiento de evento complejos *Esper*, utilizada para el desarrollo de la herramienta:

```
grammar org.xtext.example.mydsl.MyDsl2 with org.eclipse.xtext.common.Terminals

generate myDsl2 "http://www.xtext.org/example/mydsl/MyDsl2"

Domainmodel:
 (rules+=RuleParts | events+=Event)*
;

//--------------------EVENTS------------------------------
Event:
 'create' 'schema' name=ID eventattributes=Attributes ';'
;

Attributes:
 '(' attribute+=AttributesDefinition (',' attribute+=AttributesDefinition)* ')'
;

AttributesDefinition:
 name+=ID type+=ID
;
//---------------------RULES------------------------------
RuleParts:
 (nameRule = Name) (insert = Insert) (priority = Priority)? (selectRule = Select) (fromRule = From)
      (groupBy = GroupBy)? (having = Having)?';';

Insert:
 'insert' 'into' name=ID
```

```
;

Name:
 '@Name' '(' name=STRING ')'
;

Priority:
 '@Priority' '(' priorityInt = INT ')'
;

Select:
 'select' ( selectAttributes += SelectAttributesDefinition ('as' alias+=ValidID)? )+
 (',' selectAttributes += SelectAttributesDefinition ('as' alias+=ValidID)?)*
 | (asterisk?='*')
;

KindSelectAttributesDefinition:
  singleSelectDefinition = SingleSelectDefinition
 | defaultMethod = DefaultMethods
 | int = INT
 | string = STRING
;

SelectAttributesDefinition:
 rightSide += (KindSelectAttributesDefinition) (operator+=Operators leftSide+=
 (KindSelectAttributesDefinition ))*
;

SingleSelectDefinition:
 event+=[SingleDefinition] '.' (attribute+=ID | '*' )
;

From:
 'from' ((event=[Event] ('(' anything = Anything')'
 | '.' anything=Anything ))
 | pattern = Pattern )
;

//------------------Pattern------------------------------
Pattern:
 'pattern' '[' joinFollowBy=JoinFollowBy ']' ('.' win=Win)?
;


JoinFollowBy:
 followsByJoinList+=AbstractFollowBy (operator+=Operators followsByJoinList+=
 AbstractFollowBy)*
```

```
;

AbstractFollowBy:
 (=> followBy = FollowBy | '(' followBy = FollowBy ')' ) (wherePart=FollowByWhere)?
;

//------------------FollowBy------------------------------
FollowBy:
 leftSide =TerminalExpression (=> '->' rightSide+=TerminalExpression)*
;

TerminalExpression:
 every?='every' everyExpression = FollowBy
 | parenthesis?='(' betweenParenthesis = FollowBy ')'
 | singleDefinition = SingleDefinition
;

KindOfEvent: Event | Insert;

SingleDefinition :
 (=> name=ID '=')? simpleEvents=[KindOfEvent] (=>'('anything=Anything')')?
;

//--------------------------Win--------------------------
Win:
 'win' ':' defaultMethod=DefaultMethods
;

//----------------------Where-----------------------------
FollowByWhere:
 '(' FollowByWhere ')'
 | 'where' timer=Timer
;

Timer:
 'timer' ':' defaultMethod=DefaultMethods
;

//------------------GroupBy-----------------------------
GroupBy:
 'group' 'by' anything = Anything
;

//------------------Having-----------------------------

Having:
 'having' defaultMethod = DefaultMethods (operator=Operators) anything=Anything
```

```
;

//-------------------------------------------------------------

DefaultMethods:
        name=NameMethod '(' anything = Anything ')'
;

ValidID:
        ID | NameMethod
;

NameMethod:
        'avg'
        | 'current_timestamp'
        | 'count'
        | 'max'
        | 'within'
        | 'time_batch'
        | 'time'
;

//-------------------------------------------------------------

Anything:
  {Anything} =>(ID | INT | STRING | '.' | operator += Operators | extraParenthesis +=
     ExtraParenthesisRule | 'where' | ANY_OTHER)*
;

ExtraParenthesisRule:
  '(' Anything ')'
;

//-------------------------------------------------------------
enum Operators:
equal = '=' | lessThan = '<' | moreThan = '>' | lessEqualThan = '<=' | moreEqualThan = '>=' |
     and = 'and' | or = 'or' | between = 'between' | in = 'in' | not = 'not' | notIn = 'not in' | plus =
     '+' | minus = '-' | multiplication = '*' | isnot = 'is not'
;
```

Apéndice D

Conjuntos de reglas utilizados

D.1. Smart House

```
create schema Home(id String, temp Int);
create schema Person(name Sttring);

@Name("TempIncrease")
insert into TempeIncrease
select h2.ts as ts, h1.id as id, h2.temp as temp, h2.temp - h1.temp as incr
from pattern [(every (h1 = Home() ->
                      h2 = Home(h2.temp - h1.temp >= 2 and h2.id = h1.id)))
where timer:withhin(1 minutes)];

@Name("TempWarning")
insert into TempWarning
select t4.ts as ts t1.id as id, t4.temp
from pattern [(every (t1 = TempIncrease(t1.temp >= 33))
                -> (t2 = TempIncrease(t2.temp > t1.temp and t2.id = t1.id))
                -> (t3 = TempIncrease(t3.temp > t2.temp and t3.id = t1.id))
                -> (t4 = TempIncrease(t4.temp > t3.temp and t4.id = t1.id)))
where timer: within(5 minutes)];

@Name("COHigh")
insert into COHigh
select h1.ts as ts, h1.id as id
from pattern [( every (h1 = Home(h1.co >= 5000)))];

@Name("FireWarning")
insert into FireWarning
select tw.id as id, coh.ts as ts
from pattern [(every (coh = COHigh()) ->
              every (tw = TempWarning(tw.id = coh.id)))
where timer: within(5 seconds) ];
```

```
@Name("NobodyHome")
insert into NobodyHome
select p.ts as ts, h.x as x, h.y as y, h.id as id
from pattern [(every h = Home(not dopen) -> every (p = Person(
    (p.x <= (h.x - Math.sqrt(h.sqre)/4)) or
    (p.x >= (h.x - Math.sqrt(h.sqre)/4)) or
    (p.y <= (h.y - Math.sqrt(h.sqre)/4)) or
    (p.y >= (h.y - Math.sqrt(h.sqre)/4))) ))

where timer:within(3 seconds)];

@Name("CallFireDepartment")
insert into CallFireDepartment
select fw.id as id, fw.ts as ts
from pattern [(every (nh = NobodyHome()) ->
                (fw = FireWarning(fw.id = nh.id)))
where timer:within(5 seconds)];
```

D.2. MotorBike

```
create schema Motorbike(tirePressure1 Int, tirePressure2 Int, speed Double, seat Boolean);

@Name('BlowOutTire')
context SegmentedByMotorbikeId
insert into BlowOutTire
select current_timestamp() as timestamp, a2.motorbikeId as motorbikeId,
        a1.location as location_a1,
        a1.tirePressure1 as tirePressure1_a1,
        a1.tirePressure2 as tirePressure2_a1,
        a2.location as location_a2,
        a2.tirePressure1 as tirePressure1_a2,
        a2.tirePressure2 as tirePressure2_a2
from pattern [(every a1 = Motorbike(a1.tirePressure1 >= 2.0) ->
        a2 = Motorbike(a2.tirePressure1 <= 1.2) where timer:within(5 milliseconds))
        or
        (every a1 = Motorbike(a1.tirePressure2 >= 2.0) ->
        a2 = Motorbike(a2.tirePressure2 <= 1.2) where timer:within(5 milliseconds))]);

@Name('Crash')
context SegmentedByMotorbikeId
insert into Crash
select current_timestamp() as timestamp, a2.motorbikeId as motorbikeId,
        a2.location as location, a1.speed as speed_a1, a2.speed as speed_a2
from pattern [every a1 = Motorbike(a1.speed >= 50) -> a2 = Motorbike(a2.speed = 0)
where timer:within(3 milliseconds) ];
```

```
@Name('DriverLeftSeat')
context SegmentedByMotorbikeId
insert into DriverLeftSeat
select current_timestamp() as timestamp, a2.motorbikeId as motorbikeId, a2.location as location, a1.
    seat as seat_a1, a2.seat as seat_a2
from pattern [every a1 = Motorbike(a1.seat = true) -> a2 = Motorbike(a2.seat = false)];


@Name('OccupantThrownAccident')
insert into OccupantThrownAccident
select current_timestamp() as timestamp, a3.motorbikeId as motorbikeId, a3.location as location
from pattern [every-distinct(a1.motorbikeId, a1.timestamp) a1 = BlowOutTire ->
        (a2 = Crash(a1.motorbikeId = a2.motorbikeId) ->
        a3 = DriverLeftSeat(a1.motorbikeId = a3.motorbikeId)) where timer:within(3 milliseconds)];

@Name('AccidentsReport')
select current_timestamp() as timestamp, a1.location as location, count(*) as count
from pattern [every a1 = OccupantThrownAccident].win:time_batch(86400 milliseconds)
group by a1.location;
```

D.3. Nuclear power station

```
create schema TemperatureEvent(timestamp String, value Float);

@Name("Monitor")
insert into Monitor
select avg(a1.value) as avg_temp
from TemperatureEvent.win:time_batch(10 seconds) a1;

@Name("Warning")
insert into Warning
select a2.timestamp as timestamp, a1.value as temp1, a2.value as temp2
from pattern [(every (a1 = TemperatureEvent(a1.value > 400)
-> a2 = TemperatureEvent(a2.value > 400)))];

@Name("Critical")
insert into Critical
select a4.*
from pattern [(every (a1 = TemperatureEvent(a1.value > 100)
-> a2 = TemperatureEvent(a2.value > a1.value)
-> a3 = TemperatureEvent(a3.value > a2.value)
-> a4 = TemperatureEvent((a4.value > a3.value
and a4.value > (a1.value * 1.5))))) ];
```

D.4. Air Quality

Debido a que este conjunto de reglas es bastante extenso se remite al lector al siguiente enlace: `http://atenea.lcc.uma.es/Descargas/CEP/AirQuality.zip` donde podrá descargar el proyecto que contiene un fichero en texto plano (`air-quality-event-patterns.txt`) donde se encuentran este conjunto de reglas.

Bibliografía

[ANTLR, 2018] ANTLR (2018). Antlrworks: The antlr gui development environment. http://www.antlr3.org/works/. Accedido el 09 de Junio de 2018.

[Atenea, 2018] Atenea (2018). Grupo de investigación. http://atenea.lcc.uma.es/. Accedido el 31 de Mayo de 2018.

[Booch, 2002] Booch, G. (2002). Growing the uml. *Software and Systems Modeling*, 1(2):157–160.

[Boubeta-Puig, 2018] Boubeta-Puig, J. (2018). Medit4cep: Una solución dirigida por modelos para el procesamiento de eventos complejos en soa 2.0. http://rodin.uca.es/xmlui/handle/10498/18773. Accedido el 20 de Junio de 2018.

[Brambilla et al., 2017] Brambilla, M., Cabot, J., and Wimmer, M. (2017). Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 3(1):1–207.

[Burgueño et al., 2018] Burgueño, L., Boubeta-Puig, J., and Vallecillo, A. (2018). Formalizing complex event processing systems in maude. *IEEE Access*.

[Cabot and Kolovos, 2016] Cabot, J. and Kolovos, D. S. (2016). Human factors in the adoption of model-driven engineering: an educator's perspective. In *International Conference on Conceptual Modeling*, pages 207–217. Springer.

[Cugola and Margara, 2012] Cugola, G. and Margara, A. (2012). Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15.

[Dick and Ceriel, 1990] Dick, G. and Ceriel, H. (1990). Parsing techniques, a practical guide. Technical report, Technical Report, Tech. Rep.

[Eclipse, 2018a] Eclipse (2018a). Desktop ides. https://www.eclipse.org/ide/. Accedido el 1 de Junio de 2018.

[Eclipse, 2018b] Eclipse (2018b). Eclipse modeling framework. https://www.eclipse.org/modeling/emf/. Accedido el 09 de Junio de 2018.

[Eclipse, 2018c] Eclipse (2018c). Gef (graphical editing framework). https://www.eclipse.org/gef/. Accedido el 09 de Junio de 2018.

[Eclipse, 2018d] Eclipse (2018d). Xtend. https://www.eclipse.org/xtend/index.html. Accedido el 17 de Junio de 2018.

[EsperTech, 2018] EsperTech (2018). Complex event processing streaming analytics. http://www.espertech.com/esper/. Accedido el 31 de Mayo de 2018.

[Etzion and Niblett, 2010] Etzion, O. and Niblett, P. (2010). *Event Processing in Action*. Manning Publications Company.

[Eugster et al., 2003] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131.

[Gallardo, 2018] Gallardo, D. (2018). Fundamentos de las metodologías ágiles. https://es.slideshare.net/domingo.gallardo/fundamentos-de-las-metodologiasagiles. Accedido el 01 de Junio de 2018.

[García et al., 2013] García, J., García, F., Pelechano, V., Vallecillo, A., Vara, J., and Vicente-Chicote, C. (2013). Desarrollo de software dirigido por modelos: conceptos, métodos y herramientas. *Ra-ma Editorial*.

[Graphviz, 2018a] Graphviz (2018a). Dot guide. http://www.graphviz.org/pdf/dotguide.pdf. Accedido el 09 de Junio de 2018.

[Graphviz, 2018b] Graphviz (2018b). The dot language. http://www.graphviz.org/doc/info/lang.html. Accedido el 02 de Junio de 2018.

[Grimaldi, 1997] Grimaldi, R. P. (1997). *Matemáticas Discretas y combinatoria*. Addison Wesly.

[McEwen and Cassimally, 2013] McEwen, A. and Cassimally, H. (2013). *Designing the internet of things*. John Wiley & Sons.

[Microsoft, 2018] Microsoft (2018). Access sql: conceptos básicos, vocabulario y sintaxis. https://support.office.com/es-es/article/access-sql-conceptos-b%C3%A1sicos-vocabulario-y-sintaxis-444d0303-cde1-424e-9a74-e8dc3e460671. Accedido el 02 de Junio de 2018.

[Ministerio de Educación, 2018] Ministerio de Educación, C. y. D. (2018). Becas de colaboración. https://www.mecd.gob.es/mecd/servicios-al-ciudadano-mecd/catalogo/general/educacion/998142/ficha.html. Accedido el 31 de Mayo de 2018.

[Moreno et al., 2018] Moreno, N., Bertoa, M. F., Barquero, G., Burgueño, L., Troya, J., García-López, A., and Vallecillo, A. (2018). Managing uncertain complex events in web of things applications. In *International Conference on Web Engineering*, pages 349–357. Springer.

[Mozilla, 2018] Mozilla (2018). Cómo funciona css. https://developer.mozilla.org/es/docs/Learn/CSS/Introduction_to_CSS/Como_funciona_CSS. Accedido el 02 de Junio de 2018.

[OMG, 2018] OMG (2018). Unified modeling language. http://www.uml.org/. Accedido el 02 de Junio de 2018.

[proyectosagiles.org, 2018] proyectosagiles.org (2018). Desarrollo iterativo e incremental. https://proyectosagiles.org/desarrollo-iterativo-incremental/. Accedido el 01 de Junio de 2018.

[Selic, 2008] Selic, B. (2008). Manifestaciones sobre mda. *Novática: Revista de la Asociación de Técnicos de Informática*, page 13.

[Vallecillo, 2018] Vallecillo, A. (2018). Los 'planos' del software. https://elpais.com/tecnologia/2016/11/22/actualidad/1479834209_075442.html. Accedido el 01 de Junio de 2018.

[Xtext, 2018] Xtext (2018). Xtext language engineering. https://www.eclipse.org/Xtext/. Accedido el 31 de Mayo de 2018.